Lecture 2: How Software Breaks (Web)
Ayesha Ahmad

Summary: This lecture discusses how software weaknesses on the Web manifest, the vulnerabilities that lead to these weaknesses being exploited, and how they can be mitigated. The topics examined include the concept of weird machines, how privilege escalation works on the web via injection attacks such as Cross-Site Scripting, and defensive principles and programming techniques that can be used to guard against these injection attacks. Cross Site Request Forgery is given as an example of another type of attack, that is, a web session attack. Finally, two attack proxy tools are introduced; Burp Suite and Zaproxy. These tools are useful in performing web security testing of applications.

Exploiting Software
In general any software system has a set of valid inputs that generates a set of expected outputs. These predictable inputs and outputs form the expected behaviour of the system. There may also exist a set of inputs that cause the software to behave in unexpected ways that were not intended by the creator of the software as shown in Figure 1. These behaviours might be the result of exploitable bugs in the system. If an attacker can take advantage of these exploitable bugs and cause a system to execute in an unexpected way by giving it specially formed input (also known as an exploit) then this system is called a 'weird machine' [3]. Such a system is said to be programmed by the attacker using exploits. For example, a Return Oriented Programming attack forces a vulnerable program to perform a computation by chaining together gadgets (code snippets) from the original program code according to the attacker’s wish using binary exploitation via the program stack [1].

![Figure 1: Expected and unexpected system functionality.](image)

An attacker’s aim is to find out the set of inputs that produce the unexpected functionality, this set of inputs is considered as the instruction set for a weird machine. For native applications the instruction set is binary data that is executed on the processor of the machine, for web applications the instruction set is data that is interpreted as code once it has crossed the boundary into the web application. This code that has been ‘injected’ into the system may be executed at the server-side or client-side. The attack vectors for native and web applications are
different. For native applications the attack vector is binary executable data that is input to the compiled program. The attacker needs to know the memory addresses or memory layout of the target environment in order to complete an exploit. On the Web, the attack vector is data that may be JavaScript, SQL, XML or any another scripting or markup that can be evaluated as code. Table 1 summarises the differences between native and web applications with respect to exploits.

<table>
<thead>
<tr>
<th></th>
<th>Native App</th>
<th>Web App</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Vector</td>
<td>Binary data run as code</td>
<td>High-level language or data that can be evaluated as code</td>
</tr>
<tr>
<td>Executed by</td>
<td>Executed on lowest-level execution environment, the processor</td>
<td>Interpreter executes code</td>
</tr>
<tr>
<td>Execution Environment</td>
<td>Executes on processor of target machine</td>
<td>Executes on server or client browser</td>
</tr>
<tr>
<td>Type of exploit</td>
<td>Binary exploitation, e.g. buffer overflows</td>
<td>Code injection, web session attacks</td>
</tr>
</tbody>
</table>

Table 1: Differences between Native and Web application exploits.

Privilege Escalation on the Web
Privilege escalation takes place when a user (or attacker) is able to carry out actions in a system that they are not authorised to perform. Such a user has effectively increased their privilege level with respect to that system and could potentially perform harmful actions like stealing the user’s data or downloading malicious content. Privilege escalation can occur when a vulnerability in the system is exploited where the vulnerability could be the result of an implementation flaw or design flaw. In the web application domain, privilege escalation is said to have occurred when an attacker is able to have their script executed by other users [7]. This is accomplished via an injection attack such as Cross-site scripting (XSS) or SQL injection which will be described in detail later.

Current Web applications are composed of dynamic web pages that are built with a combination of markup (HTML, CSS) and high-level scripting (JavaScript) languages, and data is frequently exchanged using JSON and XML formats. In addition to this, on the server-side SQL code is often executed to query databases to bring dynamic content to the web page. This means that an injection attack on the Web usually involves code written in a language such as JavaScript or SQL, and even JSON and XML can be evaluated as code. Just as valid code in a Web application is run on the server or in the client’s browser, malicious code can also execute on both the client and server. SQL injections target the server and are executed by the server, in addition JavaScript objects received from the client can also be evaluated by the server. JavaScript injections target the client as they are executed by the browser. The server is still protected from attacks involving buffer overflows as the server executes high-level interpreted languages
such as Python and PHP [2]. Two types of injection attacks are described in the following section.

**Injection Attacks**

**Cross-Site Scripting:** The *same origin policy* is an important principle in Web application security whereby browsers ensure that only script originating from the same source is executed in a *browser* window [8]. The same source means the script and data must come from the same URI scheme, host and port number. In a Cross-site scripting (XSS) attack this policy is circumvented by the attacker as the attacker’s own code is run by the browser. The attacker manages to have their malicious script run by another user or users by tricking the user or browser into executing the script. The different types of XSS attacks are

- **Persistent:** In this form of attack the malicious script is stored on a server by the attacker, for example, script written into a discussion forum post, or in a stored input form. At a later time when the server serves the page containing the script to a user, the malicious script is executed by the browser. In case of a discussion forum, the attacker’s code might be run by many users when they view the forum page.

- **Reflected:** The malicious script is not stored on the web server instead the user is tricked by the attacker into clicking on a (phishing) link that contains the malicious script in the URL parameters. If the server response contains the URL parameters sent – as is often the case with error messages, search results, and filling in forms (see Figure 2) – then the malicious script will reach the user's browser and be executed there.

- **DOM-based:** This attack targets vulnerable client side JavaScript, the server never sees this code and cannot perform any defensive actions on it [4]

In both persistent and reflected XSS the malicious script seems to originate from the same site and thus it unlawfully gains the privilege to execute in the browser. (In DOM-based XSS, the malicious script is not delivered by the server instead the DOM is modified on the client-side, see [9]). Once the malicious script is executing the attacker could do a number of things like change the appearance of the web page, steal the user's cookies and data for session hijacking, or download malicious exes.

![Figure 2: Example of server repeating passed input.](image-url)
**SQL Injection:** Another type of injection attack is when an attacker passes SQL code to an application and this SQL code is appended into an SQL query in the database. In the absence of preventive measures, the malicious SQL will be executed by the server with the same privileges as the database user’s application, thus, the attacker has achieved privilege escalation for their SQL code. The same-origin policy does not apply here as the SQL is executed on the server-side. With an SQL injection the attacker can add, remove, copy and modify data, add new users (if this user has the required permissions), and in general perform any action realizable using SQL provided that the current user has enough permissions for the action [2]. The mechanism for delivering such an attack is similar to XSS in that SQL code can be passed to the server through the URL or form fields. In the next section we consider how injection attacks can be remediated.

**Programming defensively**

There are several ways in which injection attacks can be mitigated. They are broadly classified under input validation and output encoding. **Input validation** means checking the input when it arrives at the server or even in the browser. These types of checks could either try to test for all invalid inputs but this is a computationally hard task with a possibly exponential number of cases to check for. The other option is to carry out whitelisting, that is, only check if the input is correct and let it through if it is. But even whitelisting input is a difficult problem as a developer may overlook certain valid input cases, also checking an input format that is similar to a programming language would require checking what that input actually does which is complicated, false positives could occur where valid (but unforeseen) input is rejected because of too strict filters [2].

Input tainting is a technique to help developers identify possibly bad or dangerous input. This is a feature that is enabled by some programming languages for example Perl and Ruby. If taint mode is turned on in the interpreter, most external input to the application is considered tainted until it has been validated. If the application tries to use a tainted variable in certain output contexts, for example in an SQL statement, the program terminates with an error. The developer must un-taint the data by either validating it by comparison with a regular expression or explicitly setting the value [14].

For the case where data is being passed to a database as an SQL parameter, **prepared statements** could be used. In such a statement '?' placeholders are used to identify the location of SQL parameters that will provided by the web application. The rest of the statement, called the statement template, will be compiled by the database management system, it will be executed when the parameter values are sent by the web application [10]. The passed SQL will be treated as a parameter and not as executable script. An example of a prepared statement:

```
INSERT INTO ITEM(name, description) VALUES (?, ?)
```

An injection attack is successful only if the injected data is interpreted as code by the target system. To prevent this from happening **output encoding** can be performed to transform characters (and strings) with special meaning in the output context to benign data. Every output context has its own escaping rules whereby this encoding can be performed. For example, if the output context is HTML, angle brackets '<' '>' in input are transformed to '&lt;' and '&gt;'. If the output context is JavaScript, it could be fully encoded as numerical codes and in this way the JavaScript text would never appear in the source code [4]. The encoding will be different for
different contexts so output encoding can only be performed when the output context is known. Figure 3 illustrates where input validation and output encoding occur and provide some more examples of various output contexts.

An interesting notion is that input validation is not even needed if all outputs are being encoded according to the output context, in such a scenario code injection would never be successful. However an attacker may want to exploit the validation logic of a system by passing in malformed input, so in this case input validation becomes necessary. In general it is good practice to perform both input validation and output encoding in a system.

![Figure 3: Output encoding in different contexts](image)

The most reliable input validation can be performed on inputs that are well known and limited in number, or on those inputs that can be represented by a regular expression or a context-free grammar. However, in reality, these kind of input classes are rare and by and large, system developers can expect input that do not adhere to strict rules. The best practice would be to discard bad input as soon as it is detected instead of trying to understand what it means, or to restrict the types of things that can be input at the user interface. This saves on the computational resources of the system. Another good practice would be to do input validation at both the client and server-side (as an attacker could send data straight to the server via a URL which would bypass the client side checks). The inputs to a system must not be too complex, for example, if the input is similar to a programming language, then checking what the input does in this case is equivalent to having to figure out what the input program does – always prefer simple input to complex.

**Web Session Attacks**
A web session attack involves an attacker being able to force a user to perform actions against their choosing on websites on which they are currently logged in. This type of attack is called Cross-Site Request Forgery (CSRF) and the attacker can perform actions on the user's behalf on any site where the user is logged in or authenticated. These are actions that the user has not authorized. The attacker only needs to trick the user into clicking a link or going to a page set up
by the attacker, that leads to a request being generated to a server the user is in session with. Since the user is currently in a session with the server, the attacker could masquerade as the user – all requests would be sent in the session context of the user. Wherever the user is logged in, for example in shopping sites, their bank, email, the attacker can also access and buy things, transfer money, and send emails as if they were same as the user. CSRF attacks aim to change the state of the server, that is, to perform some action as described above like transferring money or sending email rather than just reading data from the server. In fact the attacker cannot read the response to the forged request because of the same-origin policy [5].

This type of attack is made possible by the stateless nature of web applications. As each web page request is separate, the server needs some extra data to be able to group requests together. This extra information is stored in the form of a session cookie. The cookie contains information to identify a client during a session with the server and the browser sends the session cookie with each request to the server. This is where an attacker can exploit a vulnerability in the system – if an attacker can make the user visit a page on a server where they are logged in, the server will believe that this request is knowingly generated by the user, and the server will process the request.

There are several ways to protect against CSRF attacks. One is the use of randomly generated tokens [5]. The basic idea is that predictable information must not be used to identify a session, instead a random token should be generated for each session and this token is not stored in any cookie. The token is used to identify the user. Since the token is randomly generated, the attacker cannot know what the token is and the attacker’s request will not be processed by the server because of the incorrect or missing token. Many web frameworks make the task of generating and maintaining tokens easy. Another technique is for web applications to request re-authentication of the user for operations that cause a state change on the server. The attacker does not know the user’s password or username and so cannot bypass this step. Randomised URLs can also be used to protect against CSRF attacks as the attacker will not be able to guess the expected URL, however this prevents users from bookmarking pages with randomised URLs.

**Attack Proxy Tools for Testing Web Security**

There are several tools available for testing security on the web. Two of these are Burp Suite [12] which is commercially available, and OWASP Zed Attack Proxy (ZAP) [13] which is freely available. An attack proxy intercepts requests between a browser and a server and allows the requests and responses to be modified. Users can modify requests through the tool and observe the results in their browser as well as in the tool. Users can ‘attack’ web applications and see what vulnerabilities exist. These tools provide a lot of functionality such as crawling web pages, automatic scanning for vulnerabilities, and performing attacks. The following description relates to features in Burp Suite although the ideas can be applied to Zaproxy as well since both the software provide similar functionality; Performing any web application testing with an attack proxy often involves the following steps (as described in Burp Suite):

1) **Recon and Analysis:** This part of Burp allows web application testers to study and analyse a given application. Using a mix of automatic and manual techniques the tester can map the application by following links, submitting forms and following the different
flow paths in a web site. Once all the information has been collected, Burp can explore
the attack surface of the application

2) **Tool Configuration:** In order to interact with target applications in different ways, the
tester may need to use different configuration settings with the attack proxy. These
configurations can control how authentication is handled, what the attack target is, how
session handling is performed, and the scheduling of tasks

3) **Vulnerability Detection and Exploitation:** Once the initial analysis and configuration
is complete, the tester can move on to discovering vulnerabilities in the application.
Burp provides several tools that to discover these vulnerabilities, namely the Scanner,
Repeater, Intruder, and Sequencer. Often a tester will not just use these tools separately,
but will use these tools together exchanging information between them, and use the
browser for testing results. Several examples of how an application can be tested with
these tools are given in [15].

References


