L2: How software breaks (web)
The lecture topics were the concept of how applications break, with examples focused on web application vulnerabilities, and the practical security testing of applications communicating over HTTP with Burp Suite.

The first part of the lecture was the theory, with different vulnerability examples. The second part consisted of demonstrating the commercial version of Burp Suite.

Background
This chapter explains the theory behind how software and applications break. The overall concepts are applicable to all kinds of software, not just web applications. Currently web applications present the far majority of the developed user facing software, so having a basic understanding on the issues affecting them is required knowledge for anyone interested in software security.

Weird machines
An application can be considered to be a "weird machine"\(^1\)\(^2\) – it has two kinds of input; valid input causing the machine to operate within the boundaries of expected functionality, and other input affecting the operation of the machine. This other input is something outside the range of valid input, and it controls the so-called weird machine (i.e. it reveals and controls unexpected functionality).

![Diagram of weird machine](image)

Whether the application is a native binary application, a web application or embedded software doesn’t matter – any program containing unexpected functionality, which can be triggered by an entity interacting with the software, fulfills the definition of a weird machine.

Modern web applications
The web applications of today contain an HTML page displaying a basic structure of the page, and the client-side features and functionalities are loaded from JavaScript libraries containing instructions for the browser. The location of the JavaScript libraries is defined on the HTML page and the browser fetches and interprets the defined JavaScript code after loading the page into the user’s browser.

The illustration below describes the high-level view for the client-side execution environment. The web application runs on the server-side, with the JavaScript implementation running on the client-side inside

---

The browser. The JavaScript execution environment is loaded from the surrounding HTML – in other words, the page contains both data and also the client-side code, which is interpreted in the JavaScript engine.

The data is often sent to and received from the server in JSON format. The JSON format is then parsed and consumed by the JavaScript code. The ability to modify the client-side execution environment can have as big an impact in the application as the ability to modify the server-side execution environment.

As mentioned earlier, the HTML code defines both the page contents and the applications “run” on the page by the browser. Mixing data and executable code is a particular problem of this concept.

In order to sandbox HTML pages into their own environments, browsers utilize a concept called same-origin policy. It restricts a random page from accessing for example online bank's web page with JavaScript – that is, only scripting content originating from the same host and port, using the same protocol, can access the data on the page. A web site hosted on example.org cannot access a website hosted on securebank.com using JavaScript, unless the Secure Bank application (securebank.com) explicitly includes those script resources from example.org.

The same origin policy is defined based on the following information:

\[ \text{protocol://host:port} \]

The ability to bypass the same-origin policy, either through a universal vulnerability in the user’s browser or in the web application itself opens up numerous opportunities for an attacker.

**Common web vulnerabilities**

An input validation vulnerability occurs when the software is interpreting supplied input as code. This enables an attacker to give malicious input to the application, and use it to program the weird machine. In other words, the attacker changes the behavior of the program by giving it input, which the program erroneously interprets as commands.

The first three of the following examples are input validation vulnerabilities (reflected and stored XSS, SQL injection). The last one is an example of a confused deputy problem.

---


**Reflected cross-site scripting**

Cross-site scripting (also known as XSS) occurs when an application writes user input into the DOM "as is". This offers an attacker the possibility to modify the DOM structure and make the user's browser rendering the HTML page execute scripting content defined by the attacker. Usually this scripting content is JavaScript. Injecting HTML onto a page and running arbitrary scripts within the browsing context of the victim allows the attacker to bypass the same-origin policy and control the victim's browsing session on the target site.

A reflected XSS attack can be done with GET or POST requests. The following conditions must be fulfilled for an attack to work:

1. the attacker must have access to the application at some point to first discover an exploitable XSS vulnerability - a parameter containing the attacker injection and payload must end up on the page "as is".
2. the attacker must have a method for targeting application users, through e-mail, IM or messages boards, which easily allow an unsuspecting victim to click on a link.
3. the victim must click on the malicious link.

Modern web browsers contain built-in XSS filters, which can block many reflected XSS attacks and mitigate the impact of the vulnerability.

**Stored cross-site scripting**

Stored XSS is a more serious version of the same vulnerability, as the application itself stores and serves the attacker payload. Detecting this kind of application behavior is a much more difficult problem to solve.

Stored XSS attack in a nutshell:

1. Application has a feature, which displays user input "as is".
2. The user input is available to other users.
3. An attacker stores the injection and the payload using the feature.
4. A victim visits the feature displaying the malicious content, browser interprets the attacker injection and payload activates.

**SQL injection**

A structured query language injection, aka SQLi, allows an attacker to modify a database query performed by the application. Exploitation of this vulnerability requires an attacker only to have access to the vulnerable feature, and no user interaction is required.

Let's assume a web shop displays product information using the following URL structure:

```
http://example.org/shop/details.php?id=42
```

If the application then reads the "id" parameter and includes it in a SQL query using string concatenation, a normal request to the URL would look like the following:

```
SELECT * from prices WHERE product = '42'
```

An attacker is able to modify the query by injecting an apostrophe into the query and modifying the rest of the query as demonstrated by the example URL below:

```
http://example.org/shop/details.php?id=42'+OR+1=1--
```
This will result in the following SQL query:

```
SELECT * from prices WHERE product = '42' OR 1=1--
```

The modified query returns all rows from the prices tables, as the OR 1=1 equals for all WHERE comparisons. The characters "--" are used to comment out the rest of the query.

**Cross-site request forgery**

By default web browsers or web applications do not restrict a web site from making a request to another site through user's browser. For example, it is possible for example.org to reference images on imgur.com. The browser will gladly request those images and display on the example.org's page.

This behavior can be leveraged by an attacker to perform transactions on a site, which does not prevent cross-site requests by verifying a request's origin. An attacker can create either a form or a resource request (like the aforementioned image reference) to target the vulnerable site.

An example URL demonstrating a malicious image reference targeting a vulnerable feature on victim.example.org:

```
<img src=http://victim.example.org/bank/transfer.do?to_account=RU88888888&sum=500>
```

If the attacker is able to get a logged-in user to load a HTML resource containing that image reference, user's browser will send the request for the image and at the same time perform the transaction.

**Secure handling of user input**

In order to prevent an application from interpreting attacker input as code, three key strategies exist for secure handling of user input:

1. **Restrict input**
   a. Limit the length and format to the strictest definition possible

2. **Taint user submitted data**
   a. Either use the designated mode of the development environment, or use a variable naming convention, which describes the data to be untrusted until a predefined "security gate" has been passed (e.g. regular expression based validation)

3. **Perform output encoding**
   a. Use output encoding only when you know the exact operating environment. Blind output encoding can cause in other bugs, and might not protect all interconnected systems.

Encoding should only be performed when the target environment is known. Otherwise the stored input is easily mangled when transferred to another environment, or insufficient encoding is performed.

In SQL environment prepared statements can be used.
Attacking web applications

This chapter describes how web applications can be attacked in practice. For further reading on the topic, Dafydd Stuttard's The Web Application Hacker's Handbook is perhaps the most comprehensive and refined work in the field. While there are free resources (such as the OWASP documentation) available on the Internet, the aforementioned book is a good investment for anyone wanting to acquire web application security testing skills. As with all technical skills, these can only be learned in practice, as devil is in the details, when it comes to web application security. What you though you knew about web applications and browsers will change along course of progressing in the field.

Legal disclaimer

Security testing an application without an authorization from a person with authority to grant one can be interpreted as an attempt to break in to the system. Under the Finnish law, an attempt is as punishable as a successful breach. Unsolicited "penetration test" can result in criminal charges and monetary damages – test only systems owned and controlled by you, or by someone who has granted you a permission (in written).

Protocol basics

HTTP, or Hyper Text Transfer Protocol, is a stateless text based protocol used for client-server communication in the World Wide Web. The encrypted version, HTTPS, is transferred over an encrypted transport layer – typically TLS (transport layer security) or SSL (secure socket layer). While HTTP is most commonly known as being the transport protocol between servers and browsers on the web, it is also used for example as the communication protocol for REST APIs or WebServices.

The protocol supports different kinds of methods, which are used in the requests. The RFC 2616\(^5\) defines the protocol, methods, responses, and how they are supposed to be used. It must be noted that nothing prevents an application from deviating from the specification, and many of them do so. The modern browsers also contain quite a many interoperability quirks or well-meaning, but ambiguously defined, security settings, which may have unintended outcomes with web application security.

The above text contains a HTTP POST request targeting a URI/URL on example.org.

Attacker proxies

Several kinds of tools can be used for analyzing the security of web applications – the most common ones are interactive proxies, which allow a user to intercept and modify both outgoing HTTP requests and incoming request. Two different attack proxies presented during the lecture were OWASP Zap proxy and Burp Suite (Professional). Of these two, the Burp Suite is the de-facto standard in the security industry.

\(^5\) http://www.ietf.org/rfc/rfc2616.txt
There is both a free version and a commercial version available of Burp Suite. The license price (under 300 € / year) makes it affordable even to small companies – pricing for comparable security tools start at several thousands of Euros and easily reach into the tens of thousands. Quality-wise, Burp Suite is comparable to many commercial, considerably pricier, products and has fared reasonable well in different web application security testing tool comparisons.

The most interesting features of Burp Suite Professional are the interactive proxy and its history, repeater, intruder and scanner.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Example use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive proxy</td>
<td>Intercept and modify HTTP traffic</td>
<td>Modify an individual request sent from the browser or modify the received response.</td>
</tr>
<tr>
<td>Proxy history</td>
<td>List all requests and responses</td>
<td>Investigate what happens on the HTTP level when a feature is used or a transaction is performed.</td>
</tr>
<tr>
<td>Repeater</td>
<td>Repeat a single request in an editor view and display response</td>
<td>Modify a request manually and investigate if and how an application's behavior changes</td>
</tr>
<tr>
<td>Intruder</td>
<td>Create and send multiple requests according to user defined rules</td>
<td>Send a predefined list of fuzz strings in a certain parameter to the application</td>
</tr>
<tr>
<td>Scanner</td>
<td>Web application security scanner; contains both passive and active security checks</td>
<td>Select a single HTTP request to be used as a basis for running automated security tests.</td>
</tr>
</tbody>
</table>

Burp Suite supports a multitude of use cases and for example it is possible to use upstream proxies to route the traffic for example through a jump host or Tor network. Tor network can be used for hiding the source of the traffic and/or appearing to originate from another country. Extra care should be paid and operational security⁶ practiced in operations conducted over Tor network to maintain anonymity. Further details are outside the scope of this document – but they can be summarized in one simple lesson; create a new identity for performing any anonymous activities online and make sure your anonymous digital persona cannot be linked back to your real identity.

**Using an interactive proxy**

To use an interactive proxy, a browser is needed. For security testing, it is usually preferred to utilize a dedicated browser for security testing if using Microsoft Windows, as configuring the system proxy to Burp Suite (or any other proxy) may result in unwanted system traffic going through the proxy and cluttering both the proxy history and intercept mode. The same applies to other operating systems as well, when using a system proxy setting, or there is a need to access non-security testing related resources during the assignment.

---

The screenshot below shows Firefox's proxy configuration with Burp Suite:

![Firefox Proxy Configuration](image)

When a web site uses HTTPS, Burp Suite replaces the original SSL certificate with one created on-the-fly. A Burp user can either configure Burp CA as one of the trusted CAs or accept individual certificates on a case-by-case basis.

The scanner configuration can be modified to better target an application or focus on certain specific issues. The screenshots below demonstrate different configuration options for the commercial scanner.

![Commercial Scanner Configuration Options](image)

The default values are suitable for most common use cases, especially if the user is a beginner in web application security testing.
**Summary**
Most software breaks when you approach the system from a totally different perspective compared to designer's point-of-view. Assumptions have usually been made during the development process, and instead of a well defined system, most software resembles a weird machine.

The mission, should you choose to accept, is to program the said weird machine to do your bidding. In the web application world, Burp Suite (Professional) is the de-facto "integrated programming environment" for this.

**Further reading**
Sergey Bratus et al, From Buffer Overflows to "Weird Machines" and Theory of Computing

Halvar Flake, Exploitation and state machines – Programming the weird machine revisited

Mozilla Developer Network, Same-origin policy

Norm Hardy, The Confused Deputy
http://www.cap-lore.com/CapTheory/ConfusedDeputy.html

IETF Networking Working Group, Request for comments: 2616 – Hypertext Transfer Protocol
http://www.ietf.org/rfc/rfc2616.txt

Wikipedia, Operations security