GIPHouse: Outsource management
Goal 4: Security review of the OPUS project

Martijn Sprengers, Adam Cornelissen, Benjamin Mader
(GiPHouse Outsource Management)
oum@giphouse.nl
Spring 2010

July 6, 2010

Abstract

This document describes the several security aspects regarding the Open University Systems (OPUS) project. The OPUS project deploys a fully operational student registration and management software package. In this document several dimensions of software security are described and applied to the project. It also contains a security review of the code and the modules, together with an advice on how a secure system should be set up, especially for the OPUS local system administrators. It should be taken into account that there will always be a trade off between usability/performance on one hand and security on the other.

1 Introduction

1.1 About OPUS

OPUS-College (http://www.opuscollege.net) is an Academic Registration and Information System under development, based on open source products, and is meant for the registration and management of students, staff and study data within a university or other institute of higher learning. The system currently is being developed for the Institutes of Higher Learning in Mozambique, by the Information Center of Radboud University (RU), The Netherlands. The development of OPUS-College is part of a cooperation project between Dutch and Mozambican universities, entitled ICT Capacity Building at Institutes of Higher Learning in Mozambique. The project is financed by NUFFIC, the Dutch governmental organization which promotes and fosters cooperation between Dutch universities and universities in developing countries.

1.2 Goals

This research document describes the results of our security review of the OPUS system. While a thorough security check requires more time than we can spend at this time the system should not be considered secure if all found flaws are fixed; it is certainly possible that there exist (exploitable) vulnerabilities which we missed.

1.3 Scope

Due to time constraints it is not possible to do a whole thorough security review of the whole OPUS system. We constrain ourselves to looking at the top 5 flaws of the OWASP Top 10 Web Application Security Risks for 2010:

1. **Injection**
2. **Cross-Site Scripting (XSS)**
3. **Broken Authentication and Session Management**
4. **Insecure Direct Object References**
5. **Cross-Site Request Forgery (CSRF)**
6. Security Misconfiguration
7. Insecure Cryptographic Storage
8. Failure to Restrict URL Access
9. Insufficient Transport Layer Protection
10. Unvalidated Redirects and Forwards

'Injection' refers to the injection of unintended information (such as code) which may lead to security problems. This also includes the much used 'SQL Injection attack' in which SQL code is injected and executed in the web system.

Cross-site scripting (XSS)' is a type of computer security vulnerability typically found in web applications that enables malicious attackers to inject client-side script into web pages viewed by other users. An exploited cross-site scripting vulnerability can be used by attackers to bypass access controls such as the same origin policy. The impact may range from a petty nuisance to a significant security risk, depending on the sensitivity of the data handled by the vulnerable site, and the nature of any security mitigations implemented by the site's owner.

In 'Broken Authentication and Session Management' we look at how sessions are started, maintained and ended. This includes web application procedures such as logging in and logging out of OPUS.

A Direct Object Reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. An attacker can manipulate direct object references to access other objects without authorization, unless an access control check is in place.

Cross-Site Request Forgery is a type of malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.

Not in scope are things like; Defence against Distributed Denial of Service (DDOS) attacks, (mis)configuration of hardware, training of personnel, etc. The research does also not include an elaborate manual source code inspection.

1.4 Methodology

We use tools to point us to common flaws. The tools we used are:

1. WebScarab
2. Tamper Data
3. Fortify (https://opensource.fortify.com/teamserver/faq.fhtml)

In addition to these tools, we are also going to review specific parts of the code manually. Parts of particular interest are:

- Session and authentication controllers
- Self-made security handlers
- Logging functionality

Update: There was no time for an extensive manual code inspection, but we did take a look at the most important properties. We also used some automated tools, like WebScarab to check whether security conditions where violated.

2 Basic principles of computer security

Before getting into detail it is necessary to describe the basic principles of computer security. This also increases the understanding of importance of security.

2.1 Security Dimensions

Security specialists have held several concepts as the core principles throughout the years. They are described below (as obtained from [**?**]):

Confidentiality This term is used to prevent the disclosure of information to unauthorized individuals or systems. For example, a credit card transaction on the Internet requires the credit card number to be transmitted from the buyer to the merchant and from the merchant to a transaction processing network. The system attempts to enforce confidentiality by encrypting the card number during transmission, by limiting the places where it might appear (in databases, log files, backups, printed receipts, and so on), and by restricting access to the places where it is stored. If an unauthorized party obtains the card number in any way, a breach of confidentiality has occurred. Confidentially is more often and faster breached than one might think: if someone looks over your shoulder when you type in a password for example. A breach of confidentiality can take many forms, from a stolen laptop to mathematical advances in breaking cryptographic schemes.

Integrity Integrity is a concept that ensures that data or information cannot be modified without authorization. Integrity is violated when someone with wrong intentions changes data. Integrity can also be violated without some malicious intent, for example when an employee types in a wrong address or when an optical character recognition algorithm does not work properly.

Availability For any information system to serve its purpose, the information must be available when it is needed. This means that the computing systems used to store and process the information, the security controls used to protect it, and the communication channels used to access it must be functioning correctly. Availability can not only be affected on the software layer, for example by hackers attempting a denial service attack (DOS-attack), but also on the hardware layer, for example power failure, loose connectors and old modem hardware.

Authenticity In information security it is necessary to ensure that the data, transactions, communications or documents (electronic or physical) are genuine. It is also important for authenticity to validate that both parties involved are who they claim they are. This can be achieved by digital signatures and certificates from third parties.

Non-repudiation Non-repudiation ensures that some party cannot deny an action or transaction in a (computer) system. This is especially necessary for systems with a lot of transactions and sensitive information. Non-repudiation can be achieved by digital signatures and encryption, but also via non-editable log files for example.

2.2 Risk management and security controls

For a system or software package to be secure, it is also needed to know what the risks and vulnerabilities in the specific environment are. In a risk analysis one identifies the threats, assets, vulnerabilities and policies of an organization, and in this case a software product. The management of an organization or a system has several options for any given risk:

Accept it If the value of the involved asset is relatively low, a company can choose to accept the risk and it's costs.

Transfer it The costs of a risk are shared or transferred to a third party, for example an insurance company.

Deny it Some risks have a very small chance of occurrence. In this case one can decide to deny the risk, which can be very dangerous.

Mitigate it In order to mitigate risks, the organization should take appropriate control measures (which will be described further on) to reduce them.

In this study we will only focus on the last option: how to mitigate risks.To ensure this, an organization can choose of three different types of controls. They are described below:

Administrative This type of control consists not only of (written) procedures, policies, standards and guidelines within an organization, but also of laws and regulations from governments. These controls inform people how to handle in different situations and what is

needed to get something done. Administrative controls form the basis for the implementation of logical and physical controls.

Physical Physical controls ensures that access to the environment is restricted and monitored. Examples of physical controls are: (door) locks, camera's, security personnel, etc. However, separation of duties and separation of work places are also important physical security controls.

Logical Logical controls, also technical controls, only exist in the software layer. They use all kinds of software to ensure access control, monitor network traffic and protect confidential data. Examples are: passwords, firewalls, encryption, etc.

3 The OPUS project

During our visit to the Copperbelt University we visited the server rooms and asked the network administrators about the set-up and security issues. The network and server lay-out turned out be set-up pretty well. Their standards are very high: servers are well protected and every connection goes through a configured firewall. They also have back-up servers and all physical connections between the main switches and departments are made of glass fiber. Because everything turned out to be alright we will not go in to detail about the hardware set up, we only focus on the software of the OPUS-project itself.

3.1 Issues found

Overview of the issues that we found:

1. Unlimited login attempts (no captcha)
2. POST/GET changing
3. User input is not checked or validated
4. Strong passwords are not enforced
5. No re-authentication when sensitive information is altered
6. Logging of authentication decisions
7. Hashing without salt

3.1.1 Unlimited login attempts (no captcha)

Observation: There is no limit on the number of login attempts.

Risk: An attacker might attack the login script by trying possible user/password combinations by either brute-forcing them or by using a large file containing user/password combinations which are often used (a so-called 'dictionary-attack'). With the current computer- and Internet

speeds, such attacks are no longer infeasible, especially since attackers will use software to do this automatically.

Solution: The chance for an attack relies on 'easiness' of entering a user/password combination, and on the speed of which a user/password combination can be given/checked. The first can be combated by using a 'captcha', a short sequence of numbers or a simple question which humans can easily comprehend but which is hard for a computer to do. The second can be prevented by limiting the number of possible tries a user has. One way to do this is to only allow 3 tries after which the account is blocked for a certain period of time.

Figure 1: An example of a captcha.

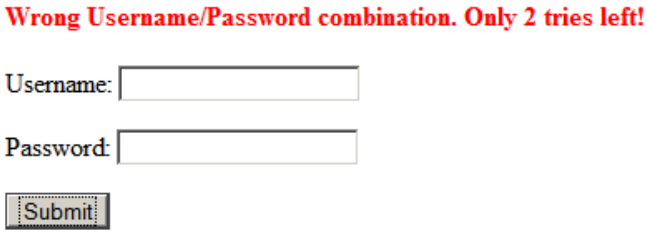

Figure 2: An example of limiting the number of login attempts.

3.1.2 POST/GET changing

Observation: Some POST/GET values can be changed causing the OPUS system to crash or behave in an unexpected way.

Risk: The flaw of being able to let the OPUS system behave in an unexpected way allows the attacker to do things he was not supposed to do. For example, you can change the hidden input fields of forms with a tool like 'Tamper Data'[1], see Figure **??**.

Solution: An attacker can _always_ change POST/GET values, even if these are in hidden fields, in drop-down lists, or in read-only marked fields. POST/GET requests are sent over the network to the website and can thus be intercepted and changed. This notion is the first step towards the solution: Check _all_ data a user can input/change.

[1] https://addons.mozilla.org/nl/firefox/addon/966/

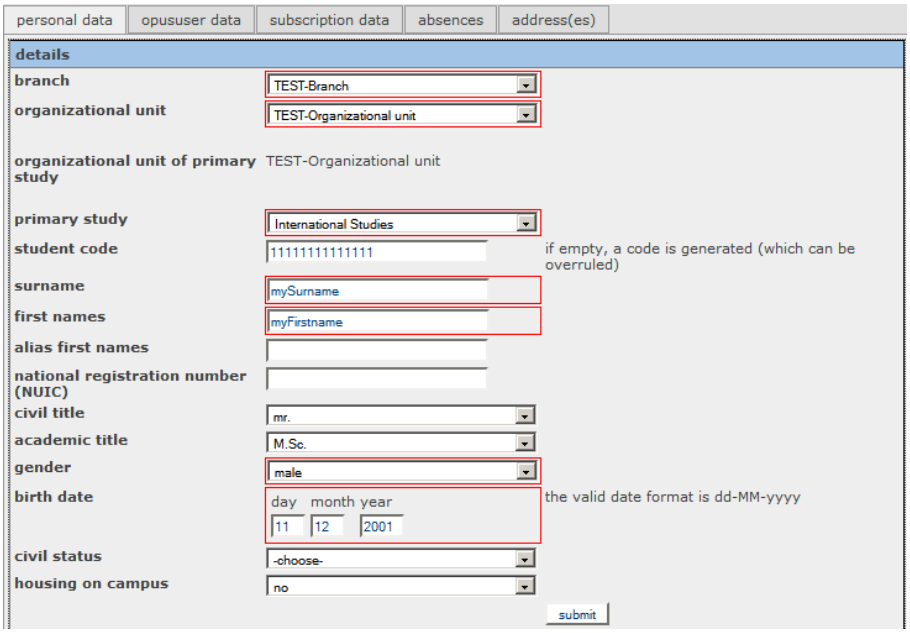

Figure 3: An example of a form in OPUS.

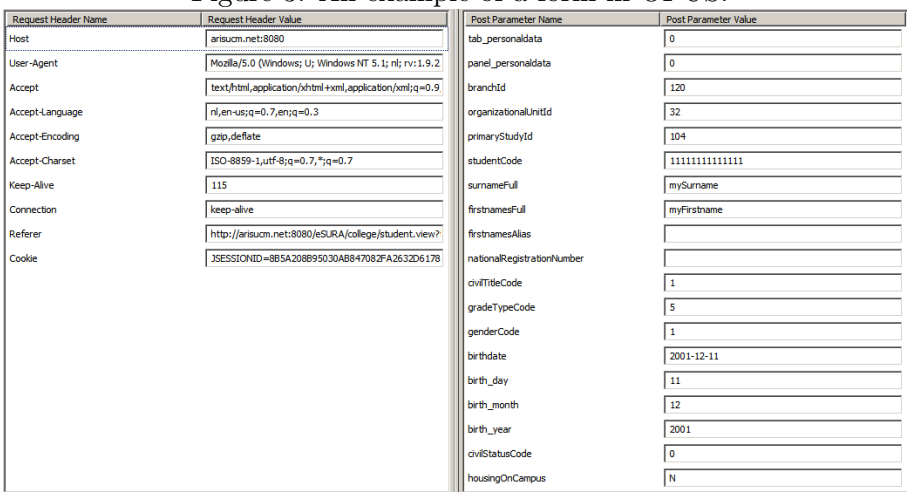

Figure 4: Possibility to change values sent to the web server (Tamper Data tool).

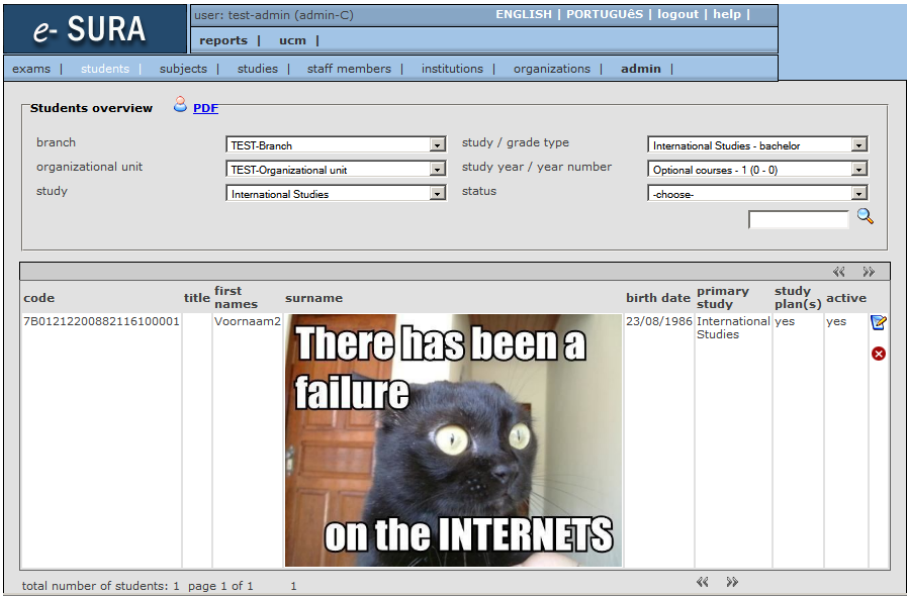

Figure 5: Command injection example (source: http://www.dmorrissey.com/).

3.1.3 User input is not checked or validated

Observation: As said before, user input is not checked. In form fields a user can write everything he wants, including HTML or SQL code.

Risk: This flaw may lead to attacks like Cross Site Scripting and Cross site request forgery. With this kind of attacks, the attacker input for example the HTML code for a meta-refresh which redirects the user's browser to another server. In Figure **??** we give an example of a what happens if user inputs the following data in the field 'Surname':

```
<img src = "http://www.dmorrissey.com/wp-content/uploads/
2010/02/lolcat-41008000x0500x375.jpg" />
```

Solution: There are easy ways to fix this. We will discuss the most successful one. This solution is based on a 'chokepoint', which is a function or a class which all user input should go through. This functions checks whether the input corresponds to the expected input and else generates an exception (based on what was expected, this could be an error or a message to the user that his input was not correct). To check the input, there are two ways to do this:

- White-listing: states which characters may be in a string
- Black-listing: states which characters may not be in a string

An implementation in Java of such a remove special characters'-function could be something like this:

```
public static string RemoveSpecialCharacters(string str)
{
```

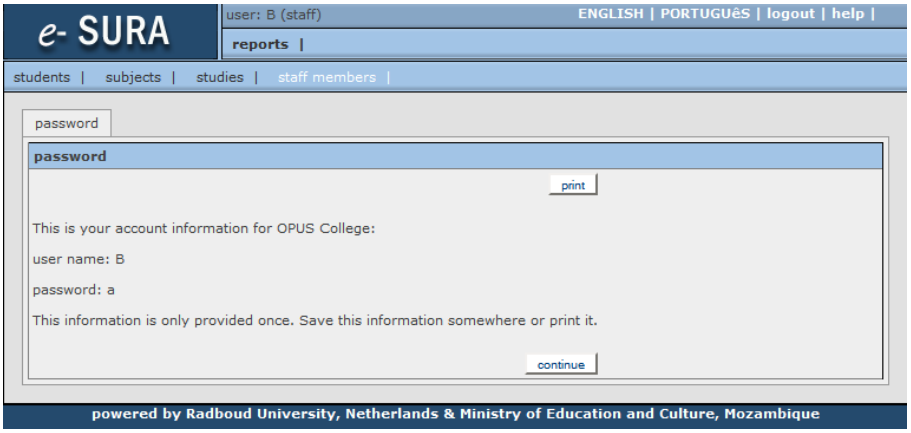

Figure 6: A user with an username and password consisting of only one character.

```
    return Regex.Replace(str, "[^a-zA-Z0-9_.]+", "", RegexOptions.Compiled);
}
```

This function uses a regular expression to filter out all the bad characters by specifying what characters are allowed (numerical and alphabetical ones). So this solution is based on white-listing.

3.1.4 Strong passwords are not enforced

Observation: When a new user logs in for the first time, he has to specify which password he wants to use. We found out that there are no restrictions on the password. A user can even have a password that only consists of one character. (see Figure **??**)

Risk: As the username and password combination is used to authenticate the user to the OPUS-system, they should be immune to brute-force attacks as mentioned in Section **??**. Now it is very easy to guess passwords by just trying.

Solution: The previous solution could be used: make a function that checks whether a password is strong or not. Criteria for a strong password can be found in Section **??**. There are several implementations available on the web that can be used. It is even better to have a password policy on the University where the system is going to be implemented. Such a policy enforces all computer users to use strong passwords.

3.1.5 Password confirmation is in plain text

Observation: When a new user logs in for the first time, he has to specify which password he wants to use. When a user enters his password, it is shown on the next screen in plain text.

Risk: A casual observer can perform a 'look-over-the-shoulder attack' when the password is presented for confirmation in plain text. (see Figure **??**). He can then use this password to login instead of the intended user.

Another risk is the injection of malicious code. Because passwords are allowed (now) to include non-alphanumeric characters such as '¡', ';'¿, and '¿' it is important that this is not executed as it could be (javascript/html/..) code which should not be executed.

Solution: In the HTML-form that is presented to the user, make three password fields like this:

- Old password: The user has to type in his current password, so the system could check it is really the user that wants to alter his password.
- New password: The new password
- Confirm new password: The new password again. Now you have to check if both the content of the 'new password' and 'confirm new password' fields are the same. This prevents the user from the 'look-over-shoulder'-attack. Important: Also check the passwords for malicious code!

3.1.6 No re-authentication when sensitive information is altered

Observation: When a user edits personal information the user does not have to authenticate himself again by giving his password.

Risk: When a user leaves his PC unattended a person walking past the PC can change personal information. Also, when by another attack a session has been hijacked it is important that sensitive information (such as a password) cannot be changed without knowing the login credentials to reduce the impact of the attack.

Solution: Have the user enter his password when he wants to change sensitive information such as his personal information or password.

3.1.7 Logging

Observation: Authentication decisions (failed logins, admin logins), hack attempts, upgrades, updates, and important changes are currently not logged.

Risk: When something goes wrong it is valuable to have a log file to determine at what point something broke. A log file can give insight to the cause of a breakdown, the way something was hacked, and who does 'bad things' to the system. A note about log files is that these should be 'append-only' meaning that it should be *impossible* to edit it (even by the administrator!) other than appending data to it. This is to prevent users from denying that they did (not) do something (non-repudiation).

Solution: Log all important decisions in an 'append-only' log file. This can be done at both the web application level, at the operating system level, or even at disk level (there exist 'append-only disks'), in increasing preference order.

3.1.8 Hashing without salt

Observation: When we did a manual code review of the session and authentication controllers, we found the following. The package 'org.uci.opus.college.web.user' contains the 'LoginController', 'LogoutController' and the 'UserSessionManager' classes. Furthermore the package 'org.uci.opus.util' contains some utility functions ('Encode.java' and 'SecurityChecker.java') that are used throughout the code. 'Encode.java' implements a MD5 hash function, but the hashes are not salted.

Risk: A hash algorithm that does not use a salt is vulnerable to an off-line brute force/dictionary attack. This attack can be performed by an attacker that could get hold of the hashes the MD5-algorithm produced.

Solution: The 'encodeMd5()' function is only used in the function 'handleRequestInternal()' in the package 'org.uci.opus.college.web.flow' in the 'StartController.java' class. As far as we could check, this function is not called. So in the current version, MD5-hashing is not used but it may be a good practice to add a salt for future use.

SHA1 can be used as an alternative to MD5. SHA1 is more secure in theory, but in real world scenarios MD5 should be sufficient.

3.2 Setting up a secure OPUS server

Since the people from the UCI will install new servers in the beginning of June 2010 (at the Copperbelt University in Kitwe, Zambia) and their level of knowledge is very high, we will not go deeper into the physical/server aspect of security. However, we could point out some important aspects that always hold to reach a sufficient security level.

3.2.1 Passwords and updates

Passwords Always choose strong passwords and never write them down (neither physical nor electronically). A strong password is defined by:

- A minimal length of 8 characters
- Use of at least one uppercase character
- Use of at least one lowercase character
- Use of at least one digit
- Use of at least one punctuation character

An example of a strong password might be: 'H@rd2Cr@k!' When we spoke with the network administrators at the Copperbelt University, it turned out that they did not use strong passwords and the passwords were written down on a paper that was attached to the servers. They assumed that no one could physically access the server room and steal the passwords, however they showed us, as guests, the server room so we were able to see the passwords. So if you write down passwords, keep them safe. Actually, don't write down passwords at all!

Updates Make sure you have the latest updates of your (server) software and especially your virus definitions. The best set up is to have the central server download the latest virus definitions and have all clients download it in their turn from the central server. When we received an USB-stick from the assistant dean of the School of Mathematics, we found out that it contained a virus. We ran it in a controlled environment to examine it. It turned out that it contained a worm (called 'Win32/Rimecud.B') and a dropper/trojan (called 'Win32/Muldrop.F'). When we confronted the network and system administrators with the virus, they said their virus definitions and software were up to date, so it could be that the terminal of the Assistant dean of the School of Mathematics was not connected to the internal University network to receive the correct virus definitions. Another thing to look after, is the adding of new systems and software. Always start from a fresh install, don't use old deamons and verify the source of the software. If you don't trust it, don't install it!

3.2.2 Setting up a secure connection (HTTPS/SSL)

As the OPUS project has a lot a of valuable and sensitive data that should be kept safe from outside observers and attackers, student grade information and payment details for instance, it is necessary to protect it well. This can be done by only granting authorized users access to the system. However, it became clear that the passwords, used to authenticate people to the system, were send in clear text together with the other credentials. It is a is a very common issue that passwords are sent in clear, since all form data is sent in clear text (this is just a property of HTTP). The password is just masked by user agents from casual observers, but it is transmitted to the server in clear text. Anyone with packet-level access to the connection can eavesdrop the data by sniffing the connection. The only way to work around this issue is to have the data encrypted. We recommended that setting up a HTTPS connection (which is basically HTTP using SSL Certificates), would be a good idea, although you can never have total security: A man-in-the-middle can fake a certificate and still get the data. This attack is the so-called "Self signed certificate". In this case, when you visit a website you will get a certificate which is signed, not by a trusted party, but by you (Opus for example) yourself. If an attacker creates a certificate signed by "0pus" (here 0 is 'zero') and replaces yours, this is hard to spot. These attacks have been done on bank websites (certificates signed by for example 'Banq of America" instead of "Bank of America" etc). So the it is necessary to buy a SSL certificate which in the first place is signed by a trusted authority and in the second place can be recognized by all well-known user agents (Internet Explorer, Mozilla Firefox, Google Chrome) automatically. The actual set up of HTTPS on the server itself is not very hard and just consists of following a few steps, so we won't go in to detail about that. See `http://tomcat.apache.org/tomcat-4.0-doc/ssl-howto.html` for more information. To be clear, this are the advantages and drawbacks of HTTPS:

Advantages

- HTTPS is proven to be very safe, there are no flaws found in the SSL/TLS protocol for over 15 years.
- It is not hard to set up. Sometimes the service provider or server host will set it up for you.
- All companies that deal with sensitive information, like banks and governments, use HTTPS to secure data transmission.

Disadvantages

- A SSL certificate costs money. Depending on what type of certificate you want: prices may vary between 30 and 1000 USD a year.
- HTTPS has some performance decrease over normal HTTP, because the data should be encrypted and the server and client have to negotiate about what cryptographic algorithms to use. In countries with slow computers and small bandwith this might me a problem. However, most computers and user agents can handle HTTPS.
- Because HTTPS is still based on HTTP, there is no concept of assured delivery. Your packets might be lost, may be delayed, anything could happen. For authentication data this should not be a problem, but if payment details get lost the transaction may be canceled or time-out.

We recommend to use HTTPS for secure data transmission with an average SSL certificate. You can always use free or self-signed certificates, but it is better to use general ones that are automatically recognized by user agents so that the user won't notice that HTTPS is used. (For self-signed and unknown certificates the browser will generate a security exception, where the user has to specify if he wants to continue.) There is one notion on encrypting data. Not all countries allow use of cryptographic tools to secure your connection. So please get information about the local rules and legislation before you install HTTPS.