TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

Secure Input for Web Applications

carried out at the

Information Systems Institute
Distributed Systems Group
and at the
Institute of Computer Aided Automation
Automation Systems Group
Vienna University of Technology

under the guidance of
Priv.Doz. Dipl.-Ing. Dr.techn. Christopher Krügel
and
Priv.Doz. Dipl.-Ing. Dr.techn. Engin Kirda

by

Martin SZYDLOWSKI
Siebenbrunnenfeldgasse 26/21/15
1050 Wien
Matr.Nr. 0025313

Vienna, 08. May 2007

Zusammenfassung

Das Web ist zu einem unerlässlichen Teil unseres Lebens geworden. Webapplikationen sind unzweifelhaft die vorherrschende Methode um Onlinedienste zu nutzen. Jeden Tag verwenden Millionen Benutzer das Web um einzukaufen, Bankgeschäfte abzuwickeln, Informationen zu suchen und zu kommunizieren. Obwohl das Web für die Benutzer den Komfort bietet, jederzeit und überall an Informationen zu kommen und Dienste zu nutzen, ist es gleichzeitig das Ziel vieler böswilliger Menschen, die sich leichten Profit von Angriffen auf ahnungslose Benutzer erwarten. In den letzten Jahren hat die Anzahl webbasierter Angriffe stark zugenommen, was die Wichtigkeit von Techniken und Werkzeugen zur Erhöhung der Sicherheit im Web unterstreicht. Zum Beispiel sind Banken, die ihre Dienste im Web anbieten, ein primäres Ziel von Phishingattacken. Weiters gab es viele Pressemeldungen über den Diebstahl von Kreditkartendaten von Millionen von Benutzern.
Ein wichtiges Forschungsproblem im Bereich der Websichertheit ist, wie man einem Benutzer auf einer unsicheren Platform (also einem Computer, der mit bösartigen Programmen infiziert ist) die sichere Übermittlung von Daten an eine Webapplikation gewährleistet. Die bisher vorgeschlagenen Lösungen benötigen in den meisten Fällen zusätzliche (und oft teure) Hardware, zum Beispiel eine Chipkarte und ein entsprechendes Lesegerät. In dieser Diplomarbeit besprechen wir Angriffe auf Webapplikationen, die über den Computer eines Benutzers ausgeführt werden können, zum Beispiel mittels eines Trojanischen Pferdes. Wir haben aktuelle Sicherheitslösungen auf Webseiten von verschiedenen Banken untersucht und präsentieren die Schwächen dieser Lösungen. Weiters stellen wir zwei neue Möglichkeiten vor, wie sichere Eingabe in Webapplikationen realisiert werden kann. Wir haben in kleinem Umfang eine Benutzerstudie durchgeführt, um zu überprüfen ob unsere vorgeschlagenen Lösungen praktikabel sind.
Da unsere Abhängigkeit vom World Wide Web weiterhin steigt, erwarten wir, dass Angriffe auf Webapplikationen seitens des Clients weiterhin zunehmen.

Abstract

The web is an indispensable part of our lives. Undoubtedly, web applications have become the most dominant way to provide access to online services. Every day, millions of users purchase items, transfer money, retrieve information and communicate over the web. Although the web is convenient for many users because it provides anytime, anywhere access to information and services, at the same time, it has also become a prime target for miscreants who attack unsuspecting web users with the aim of making an easy profit. The last years have shown a significant increase in the number of web-based attacks, highlighting the importance of techniques and tools for increasing the security of the web. For example, online banking web sites all over the world are frequent targets of phishing attempts and there has also been extensive press coverage of recent security incidences involving the loss of sensitive credit card information belonging to millions of customers.

An important web security research problem is how to enable a user on an untrusted platform (e.g., a computer that has been compromised by malware) to securely transmit information to a web application. Solutions that have been proposed to date are mostly hardware-based and require (often expensive) peripheral devices such as smart card readers and chip cards. In this thesis, we discuss some common aspects of client-side attacks (e.g., Trojan horses) against web applications. We review current protections schemes deployed on online banking sites and discover their shortcomings. Further, we present two simple techniques that can be used by web applications to enable secure user input. We also conducted a small-scale usability study to examine whether the techniques that we propose are feasible.

As our dependency on the web increases, we expect that client-side attacks against web applications will be continuing problems in the future.

Acknowledgements

First and foremost, I would like to thank my parents, for their never-ending patience and support, even when my studies took a while longer to complete than expected. I want to thank my supervisors, Engin Kirda and Christopher Krügel, for sharing their experience in scientific research with me and for bringing me back on track when I got lazy. Finally, I would like to thank all users who participated in our usability study. This work has been supported by the Austrian Science Foundation (FWF) under grants P-18764, P-18157, and P-18368.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since the advent of the web, our lives have changed irreversibly. Web applications have quickly become the most dominant way to provide access to online services. For many users, the web is easy to use and convenient because it provides anytime, anywhere access to information and services. Initially, web sites were mainly used for providing information to visitors, but today, a significant amount of business is conducted over the web, and millions of web users purchase items, transfer money, retrieve information and communicate via web applications.

Unfortunately, the success of the web and the lack of technical sophistication and understanding of many web users has also attracted miscreants who aim to make easy financial profits. The attacks these people have been been launching range from simple social engineering attempts (e.g., using phishing sites) to more sophisticated attacks that involve the installation of Trojan horses on client machines (e.g., such malicious software may be automatically installed by exploiting vulnerabilities in browsers in so-called *drive-by attacks* [35]). The strong increase in the numbers of such attacks that target web applications have highlighted the need for techniques and tools to increase the security of the web.

An important web security research problem is how to effectively enable a user using a client on an untrusted platform (i.e., a platform that may be under the control of an attacker) to securely communicate with a web application. More precisely, can we ensure the *confidentiality* and *integrity* of sensitive data that the user sends to the web application *even if* the user's platform is compromised by an attacker? Clearly, this is an important, but difficult problem.

Ensuring secure input for web applications is especially relevant for online services such as banking applications where users perform money transfers and access sensitive information such as credit card and account numbers. Although the communication between the web client (e.g., the browser) and the web application is typically encrypted using technologies such as Transport Layer Security [24] (TLS) to thwart man-in-the-middle attacks, the web client is the weakest point in the chain of communication. This is because it runs on an untrusted platform, thus, it is vulnerable to client-side attacks that are launched locally on the user's machine. For example, a Trojan horse can install itself as a browser-plugin and can then easily access, control and manipulate all sensitive information that flows through the browser. In this case, using an encryption technology such as TLS does not solve the problem. Malware that manipulates bank transactions already appears in the wild. Last year, for example, several Austrian banks were explicitly targeted by Trojan horses that were used by the miscreants to perform illegal money transactions. In most cases, the victims did not suspect anything, and the resulting financial losses were significant and alarming. Note that even if the costs of such an attack are covered by insurance companies, it can still easily harm the public image of the targeted organization and may cause immeasurable damage such as the loss of customers (who may lose confidence in the organization).

A number of solutions have been proposed to date to enable secure input on untrusted platforms for web-based applications. The majority of these solutions are hardware-based and require integrated or external peripheral devices such as smart card readers [25, 42] or mobile phones [29]. One disadvantage of such hardware-based solutions is that they impose a financial and organizational burden on users as well as service providers. First, users typically need to buy and install the necessary hardware and service providers need to install, adapt, extend and configure their services accordingly. Second, such solutions may also involve higher maintenance costs and a part of this cost typically reflects back to the users in that they are required to pay more for the services. Third, a key disadvantage of device-based solutions is that they eliminate the anytime, anywhere advantage of the web by tying users to specific computers where the devices are available and preconfigured. Also, note that the fact that a hardware device is used to enable secure input between the web client and the web application does not necessarily guarantee that the communication channel is indeed secure. These devices often require and depend on device drivers that are installed and configured on the untrusted platform. Hence, it might be possible for the attacker to modify or replace these components with tampered versions [27, 47, 48].

1.2 Contributions

In this thesis, we discuss some common aspects of client-side attacks against web applications and present two simple techniques that can be used by web applications to enable secure input, at least for a limited quantity of sensitive information (such as bank account numbers). The main advantage of our solutions is that they do not require any installation or configuration on the user's machine. Furthermore, as our prototype implementations demonstrate, our techniques are easy to implement and integrate with existing applications. Finally, in order to evaluate the feasibility of our techniques for mainstream deployment, we conducted a usability study.
The main contributions of this thesis are as follows:

- We present a technique that extends graphical input (on-screen keyboards) with CAPTCHAs [8] to protect the *integrity* of the user input even if the user platform is under the control of an automated attack program (such as a Trojan horse).
- We present a technique that makes use of confirmation tokens that are bound to the sensitive information that the user wants to transmit. This technique helps to protect the *integrity* of the user input even if the user platform is under the control of the attacker.
- We present a *usability* study that demonstrates that the two techniques we propose in this thesis are feasible in practice.
- We present details on the prototype implementations of our techniques and describe how existing web applications can integrate them.

1.3 Organization of this thesis

Chapter 2 explains the threats against web applications and outlines the type of applications most likely to be targeted. It describes the possible attacks and gives an example of a typical client-side attack. Chapter 3 is an overview of related work concerning secure input. Additionally, currently deployed solutions for protecting user input are reviewed and their shortcomings are discussed. Chapter 4 outlines the idea behind a CAPTCHA and it's current uses. Chapter 5 presents techniques developed in the course of this thesis to enable secure input for web applications. Chapter 6 discusses their implementation. Chapter 7 presents the results of the user study and and discusses limitations of this approach. Finally, Chapter 8 summarizes and concludes this thesis.

Chapter 2

Problem Definition

Web-based services cover a wide range of applications, with different levels of criticality. Non-critical applications generally do not cause damage to the user when abused by a malicious third party. Critical applications can cause substantial personal (i.e., causing a severe breach of privacy) or financial damage to the user when compromised. However, neither of these applications is likely to be targeted by an attacker for the sole purpose of damaging another person. The main incentive of the cyber-criminal is financial gain, as it is with any real-life criminal. Therefore, the web applications most likely to be targeted are online-banking services, preferably from banks with a large customer base, online payment services such as PayPal or e-commerce sites such as eBay (for some examples, see [18, 36, 45]). In short, attacks on web applications occur most often wherever the virtual equivalent of real money can be moved.
An attacker targeting a web applications has three possible avenues of attack.

1. Compromise the server
2. Compromise the communication channel (man-in-the-middle)
3. Compromise the client

Large corporations providing web services are well aware of the risks involved and have the funds and the staff to secure their web servers adequately. There have been (successful) attacks on Bank servers in the past, but these have become less common and less successful in recent times due to the heightened awareness of the service providers.
When using state-of-the-art encryption, the communication channel can be considered secure. TLS/SSL [24] encrypts data between the application layer and the transport layer. Therefore, unencrypted data exists only on the client

computer and on the application server. The data stream between client and server cannot be read or tampered with.
On the other hand, the people using a web service are, in general, no IT professionals or computer security experts. Although, in recent times, the media tried to raise the general awareness concerning security risks in the World Wide Web, people are still widely unaware or misinformed on how these risks can affect them. Many only "use" their computers and require assistance when it comes to maintenance tasks like installing/updating applications, drivers or the operating system. Therefore, many users could have difficulties detecting malicious software installed on their computers. Considering the growing sophistication of malware writes and their attack methods, even experienced users sometimes cannot prevent a malware infection.
Additionally, and considering the anytime, anywhere nature of the web people have grown accustomed to, it is likely that they will access (possibly critical) web services from computers other than their private desktop PC. It could as well be the workstation in the office or a terminal in an Internet café in the favorite vacation spot or even a friend's PDA using wireless LAN in an airport terminal. There is often no way of knowing what software might be installed on them, but, in general, people are trusting and do not assume malicious intentions unless convinced otherwise. People will often use "foreign" computers without prejudice.
Hence, the weakest link of the web application is the client, that is, the computer a person uses to access a web service and the software running on it.

2.1 Client-Side Attacks

To successfully compromise a client computer, the attackers have to install a malicious software on it. That will give them (at least partial) control of the computers actions. In order to achieve this objective, the attackers either have to trick the user into installing the software on their behalf or exploit (use a vulnerability in a program to inject and execute foreign machine code in the program's context) a service or application on the user's computer. A combination of these two methods is also possible: The user is tricked into performing an action (e.g., click on a link) that in turn triggers an exploit.
A malicious program that tricks the user by pretending to be harmless is called "Trojan horse[1]" or simply "Trojan," in reference to the ancient Greek legend. Trojan horses do not attempt to spread further. Their only goal is to stealthily install their payload on the victim's computer. A program that spreads autonomously over the network using application vulnerabilities is

referred to as “worm”. [44] The first (unintentionally malicious) worm (Morris Worm [46]) appeared in 1988 and affected the then prevalent DEC VAX and Sun systems running BSD 4, a flavor of UNIX. Recent examples of massively popular worms include W32.Blaster, W32.Sasser and W32.CodeRed which attacked Microsoft Windows machines. A sub-family of worms combines characteristics from both Trojans and classical worms. These “mass-mailing” worms propagate via e-mail, with the worm code as attachment, disguised as image, document or archive (Examples: Melissa, Anna Kournikova, ILoveYou, MyDoom). Mass-mailing worms rely on the user to open the attachment and hence continue the spreading cycle. Nimda [9], a successful worm from 2001, used multiple attack vectors (e-mail, exploits, drive-by, open network shares and backdoors from other worms) to propagate.
To be of any use to the attackers, the malware, regardless of the propagation method, has to include some kind of payload besides the spreading routines. This payload can be the application the attackers wish to install, or a small bootstrap program that, when activated, will attempt to download the main application from the network. The latter method is used where payload size is limited, for example, in worms that exploit buffer overflows. In general, the attackers will attempt to install a backdoor that gives them control over the computer and a rootkit that attempts to conceal their actions by hiding files and processes from the user’s view.
The best-known backdoor programs for the Microsoft Windows OS are Back-Orifice, Netbus and SubSeven, and their authors refer to them, quite accurately, as “remote administration tools.” Backdoor program “features” include keystroke logging, screenshot grabbing, modification of files and system settings and installation of additional software. Traditional rootkits modify the behavior of standard system tools, like the `ls` and `ps` commands in Unix or the `Explorer` and `Task Manager` in Windows. Alternatively, rootkits change the underlying functionality in the system kernel. Examples of such rootkits include Hacker Defender, FU and NT Rootkit [23]. In recent years, after the introduction of hardware-assisted virtualization techniques[2], a new generation of rootkits has emerged, with Blue Pill [41] being the prime example. These rootkits move the entire OS inside a virtual machine while acting as hypervisor, which makes them hard to detect.
Besides these powerful tools, the attackers might use something more subtle and specialized. To attack a web application, only the web browser needs to be compromised. Since modern browsers are extensible in design, this task is

[1]The term “Trojan horse” was coined by “MIT-hacker-turned-NSA-spook” Dan Edwards in the early 1970ies.

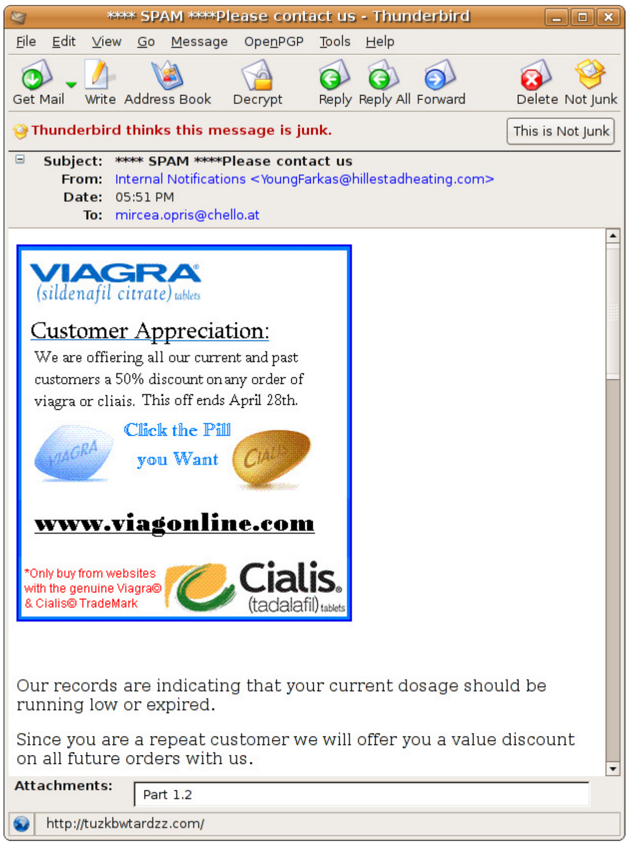
[2]VT a.k.a. Vanderpool by Intel and SVM a.k.a. Pacifica by AMD

not difficult. The Microsoft Internet Explorer supports Browser Helper Objects (BHOs), dynamically linked libraries (DLLs) that are loaded with the main application. The idea behind BHOs is to provide added functionality beyond what the browser is able to do. For instance, a BHO from Adobe Systems facilitates the display of PDF[3] files, often encountered on the web, directly inside the browser window. Unfortunately, the BHO functionality can be abused for malicious purposes. BHOs can, for example, extract and modify the content from web pages the user is viewing, or intercept and change the HTTP requests the user submits. Therefore, BHOs are the ideal means to attack web applications from the client side. Other browsers have comparable extension mechanisms. For instance, the Mozilla Firefox supports Extensions and Plug-ins that can modify it's appearance and behavior in similar fashion to BHOs. For the sake of simplicity, we will refer to all browser extensions with comparable functionality as BHOs. The following section will illustrate, with an example, the entire chain of events that constitutes a successful attack.

2.2 Trojan-Based Attack: An Example

In a typical client-side web attack, the aim of the attacker is to take over control of the user's web client in order to manipulate the client's interaction with the web application. Such an attack typically consists of three phases. In the first phase, the attacker's objective is to install malware (i.e., malicious software) on the user's computer. Once this has been successfully achieved, in the second phase, the installed malware monitors the user's interaction with the web application and waits for the user to perform a security-critical operation. The third phase starts once the malware detects that a security-critical operation is taking place and attempts to manipulate the flow of sensitive information to the web application to fulfill the attacker's objectives. A possible way for an attacker to complete phase one would be to set up a website containing an exploit for a recently discovered vulnerability in a popular browser (e.g., the Internet Explorer). For example, it could be a serious parsing-related vulnerability that allows malicious code to be injected and executed on the victim's system just by visiting the aforementioned web site. Then the attacker sends thousands of e-mails (possibly looking like Figure 2.1) containing a link to his website, and encouraging users to click on that link by offering something that might interest most of them. This technique is often used by "spammers" (people who send unsolicited e-mail

[3]Portable Document Format, an open file format for device-independent and resolution-independent 2D documents.



Thunderbird thinks this message is junk.

Subject: **** SPAM ****Please contact us
From: Internal Notifications <YoungFarkas@hillestadheating.com>
Date: 05:51 PM
To: mircea.opris@chello.at

VIAGRA
(sildenafil citrate) tablets

Customer Appreciation:
We are offiering all our current and past customers a 50% discount on any order of viagra or cliais. This off ends April 28th.

Click the Pill you Want

www.viagonline.com

*Only buy from websites with the genuine Viagra© & Cialis© TradeMark

Cialis (tadalafil) tablets

Our records are indicating that your current dosage should be running low or expired.

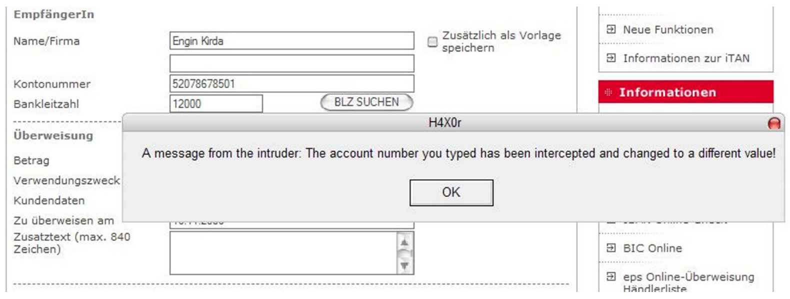
Since you are a repeat customer we will offer you a value discount on all future orders with us.

http://tuzkbwtardzz.com/

Figure 2.1: A typical SPAM e-mail. Note that the image is a hyperlink, and while the advertised URL (in the image) is `http://www.viagonline.com/`, it is actually pointing to a different URL, `http://tuzkbwtardzz.com/`. This is a common feature of malicious e-mails.

containing advertisements) who, as such, have no malicious intent, but wish to sell (mostly) legitimate goods. Since people indeed respond to spam (otherwise spamming would be unprofitable and cease to exist), we can assume that the attacker's website will be visited by some unsuspecting victims. Further, we can assume that not all users have their browser patched, even when the browser manufacturer has released a fix for the aforementioned vulnerability. As a result, a Trojan horse is automatically installed on their computers when the browser parses the contents of the web page — they have become victims of a drive-by attack.
The Trojan horse that the attacker has prepared is a Browser Helper Object (BHO) for the Internet Explorer (IE). This BHO is automatically loaded every time IE is started. With the BHO, the attacker has access to all events (i.e., interactions) and HTML components (i.e., DOM objects) within the

browser. Hence, he can easily check which web sites the user is surfing, and he can also modify the contents of web pages. In this example, the attacker is interested in web sessions with a particular bank (the Bank Austria).
Whenever an affected user is online and starts using the Bank Austria online banking web application, the Trojan browser-plugin is triggered. It then starts analyzing the contents of the bank web pages. When it detects that he is about to transfer money to another account, it silently modifies the target account number.


EmpfängerIn
Name/Firma: Engin Kirda
Kontonummer: 52078678501
Bankleitzahl: 12000
Überweisung
Betrag
Verwendungszweck
Kundendaten
Zu überweisen am
Zusatztext (max. 840 Zeichen)

H4X0r
A message from the intruder: The account number you typed has been intercepted and changed to a different value!

Figure 2.2: A BHO Trojan horse targeting the real-life online banking web application of the Austrian BA-CA bank. In this proof-of-concept attack, the user is informed of the attack with a message dialog.

Figure 2.2 depicts a screenshot of the message dialog that is displayed to a victim in a proof-of-concept implementation of a BHO-based Trojan horse that targets the real-life Bank Austria online banking web application. Obviously, in a real attack, a message dialog would not be displayed to the victim, who would not notice any suspicious behavior.
Note that the imaginary attack described here is actually very similar to the attacks that have been recently targeting Austrian banks. Clearly, there can be many technical variations of such an attack, and the possibilities of the attackers are only limited by their imagination. For example, instead of using a BHO, the attackers could also inject Dynamic Link Libraries (DLLs) into running applications or choose to intercept and manipulate Operating System (OS) calls.
The key observation here is that the online banking web application has no way to determine if the client it is interacting with has been compromised. Furthermore, if the client has indeed been compromised, all security precautions the web application can take to create a secure communication channel to the client fail. For example, the fact that an TLS web connection (i.e., an encrypted channel) is established with the client will not prevent a man-in-the-middle attack within the browser. That is, the web application

cannot determine if it is directly interacting with a user, or with a malicious application that performs illegitimate actions on behalf of a user.

2.3 Mitigating Trojan-Based Attacks

There are many points in time, where the client from the example in the previous section could have been protected from harm. First, a well-trained spam filter on the users e-mail server could have detected and dropped the attackers e-mail. Second, the e-mail application on the client computer could have built-in spam and scam protection (or use third party software for that purpose) and could warn the user of the malicious content of the e-mail. However, such software works with statistical and probabilistic methods and is not 100% reliable (e.g., see [21]).
The next points lie in the responsibility of the user. He could decide not to click on the link in the e-mail because he is aware of the dangers this might produce. Also he could be aware that in order to keep his system safe, he needs to keep his applications up to date by installing all available patches and service packs. Unfortunately, we cannot expect people to behave in a safe manner, rather we must assume that the majority will not concern themselves with such practices.
Finally, a virus scanner on the user's computer might detect the malicious software in memory or on the hard drive and remove or quarantine it. However, current anti-virus (AV) software relies on signatures[4] to positively identify malicious binaries. Virus signatures are created by AV manufacturers when a new malware is discovered and AV scanners need to update their signature database regularly to offer protection against current threats.
The conclusion is that, although there are many possibilities to protect clients from infection with malicious programs, on current platforms there is no reliable way to prevent it with absolute certainty. Combined with the fact that such and similar attacks as mentioned in 2.1 occur in great number, this implicates a high probability that some of the vulnerable clients are indeed infected. Current statistics prove this hypothesis (e.g., reports from the Windows Malicious Software Removal Tool [31]).

[4]Signature: An immutable bit pattern inside a binary which can be used to fingerprint it

2.4 Realizing Secure Input on Insecure Platforms

Following the argument from Section 2.3, a web service provider must assume that some of the clients accessing it's services are compromised. As mentioned in Section 2.1, the web server has no way to distinguish safe from unsafe clients, since the only information a server has about the client is that which the client chooses to disclose. Therefore, the server cannot single out the compromised clients and ban them from accessing it's services. This leaves the service provider with the choice of either offering a service to everyone, with the risks imposed by insecure clients, or offering no service at all. Clearly, providing no service is not an option.
When designing a secure web application, three goals have to be met. First, the application should protect the *confidentiality* of sensitive data, such as passwords, account or credit card numbers, medical records, etc. Second, it should protect the *integrity* of the data, that is, prevent unauthorized modification of the data by third parties. Finally, the application must maintain it's *usability* to remain popular. The next chapter reviews how well these goals have been met in existing applications, focusing on online-banking sites.

Chapter 3

Related Work

Client-side sensitive information theft (e.g., spyware, keyloggers, Trojan horses, etc.) is a growing problem. In fact, the Anti-Phishing Working Group has reported over 170 different types of keyloggers distributed on thousands of web sites [1]. Hence, the problem has been increasingly gaining attention among security researchers and a number of mitigation ideas have been presented to date.

Several client-side solutions have been proposed that aim to mitigate spoofed web-site-based phishing attacks. PwdHash [39] is an Internet Explorer plug-in that transparently converts a user's password into a domain-specific password so that the user can safely use the same password on multiple web sites. A side-effect of the tool is some protection from phishing attacks. Because the generated password is domain-specific, the password that is phished is not useful. SpoofGuard [16] is a plug-in solution specifically developed to mitigate phishing attacks. The plug-in looks for "phishing symptoms" such as similar sounding domain names and masked links in the web sites that are visited. Alerts are generated based on the number of symptoms that are detected. Note that both PwdHash and SpoofGuard focus on the mitigation of spoofed web-site-based phishing attacks. That is, they are vulnerable against client-side attacks as they rely on the integrity of the environment they are running in. Similarly, solutions such as the recently introduced Internet Explorer anti-phishing features [32] are ineffective when an attacker has control over the user's environment.

Spyblock [26] aims to protect user passwords against network sniffing and dictionary attacks. It proposes to use a combination of password-authenticated key exchange and SSL. Furthermore, as additional defense against pharming, cookie sniffing, and session hijacking, it proposes a form of transaction confirmation over an authenticated channel. The tool is distributed as a client-side system that consists of a browser extension that runs in an untrusted environment and an authentication agent that runs in a virtual machine environment that is "protected" from spyware. A disadvantage of Spyblock is that the user needs to install and configure it. The solutions we propose, in contrast, are server-side and require no installation or configuration by the user.

A number of hardware-based solutions have been proposed to enable secure input on untrusted platforms. Chip cards and smart card readers [25, 42], for example, are popular choices. Unfortunately, it might be possible for the attacker to circumvent such solutions if the implementations rely on untrusted components such as drivers and operating system calls (e.g., see [27, 47, 48]). As an alternative to smart-card-based solutions, several researchers have proposed using handhelds as a secure input medium [5, 29]. Note that although hardware-based solutions are useful, unfortunately, they are often expensive and have the disadvantage that they have to be installed, configured and available to users.

A popular anti-keylogger technique that is already being deployed by certain security-aware organizations are graphical keyboards. Similar to our graphical input technique, the idea is that the user types in sensitive data using a graphical keyboard. As a result, she is safe from keyloggers that record the keys that are pressed. However, there has been increasing reports of so-called "screenscrapers" that capture the user's screen and send the screenshot to a remote phishing server for later analysis [20, 43]. Also, with many graphical keyboard solutions, sensitive information can be extracted from user elements that show the entered data to provide feedback for the user. Finally, to the best of our knowledge, no graphical keyboard solution uses CAPTCHAs. Thus, the entered information can be determined in a straightforward fashion using simple OCR schemes.

Finally, client-side attacks could be mitigated if the user could easily verify the integrity of the software running on her platform. Trusted Computing (TC) [38] initiatives aim to achieve this objective by means of software and hardware. At this time, however, TC solutions largely remain prototypes that are not widely deployed in practice.

The following sections review security solutions deployed in real-life online baking applications and investigate how well these solutions protect users against client-side attacks.

3.1 Online Banking - A Security-critical Web Application

Most banks offer the following services online to their customers.

- Viewing of bank statements for all accounts and credit cards
- Payment of bills
- Transfer of funds to own or foreign accounts
- Management of investments (stocks, bonds, etc.)
- Loan application and repayment

All of the above transactions must be considered security-critical, since they operate on private and sensitive data of the customer. Most banks employ a two-tiered approach (on the client side) to protect the confidentiality and integrity of this data. The first level of protection, which aims at ensuring the *confidentiality*, is a login challenge. The customer has to provide a user ID and a password or PIN[1] to *view* the protected data (a method common on e-commerce sites). The credentials are passed to the customer either personally or via mail, but never in electronic form. It has been a long-standing practice for banks to distribute similarly sensitive data in such form (e.g., the PIN for a debit card), for practical and legal reasons. For an attacker, this limits the window of opportunity for stealing these credentials to the time when the customer needs to *input* them in order to access the bank's web services. Current methods to secure the input of confidential data are outlined in Section 3.2.

A second level of protection is implemented where the sensitive data can be *modified* (protect the *integrity*). When, for example, issuing a funds transfer, the user inputs all relevant data (target account, routing number, amount, etc.) in a web form and submits this data together with a security token. This token serves as and is equivalent to a signature *confirming* the identity of the issuer. To enhance the security, such tokens are typically single-use or valid for a limited time only. An attacker who is able to steal the token while the legitimate user enters it, cannot use it after the user has completed the transaction. There is a variety of methods for obtaining or generating such tokens, and Section 3.3 will describe the most prominent ones.

[1]Personal Identification Number

3.2 Input Methods

The keyboard is the prevalent method of inputting text, and, despite some recent advances in voice recognition, will remain so in the foreseeable future. In a modern operating system (OS) with multi-tasking and a graphical user interface (GUI), all key presses are registered by the OS. The OS makes the codes of the pressed keys available to the "foreground" application, either by actively informing the application (with a message or an interrupt) or by allowing the application to poll that information. However, the OS also provides means for applications running in the background to access the keycodes. In the Microsoft Windows operating system, the Application Programming Interface (API) call `GetAsyncKeyState(int vKey)` can be used by every running application to query the present state of the key with the keycode `vKey`. Other, more sophisticated methods to monitor keystrokes also exist (e.g., hooking the keyboard input handler). These methods have legitimate uses. For instance, a music player could allow to pause playback without the need to find and bring the application window to the foreground ("global hotkeys"). However, this functionality can be used to implement a keystroke logger without much effort. Backdoor Trojans can, and in most cases do, include keystroke logging and reporting facilities.

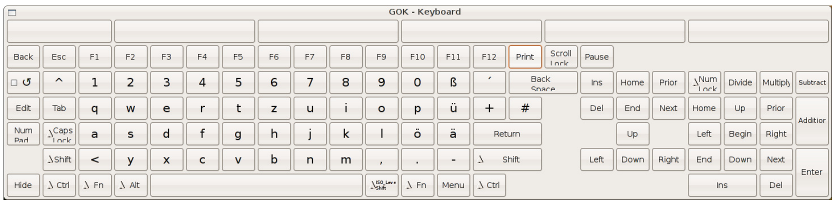


Figure 3.1: The Gnome On-screen Keyboard on Ubuntu Linux.

A possibility to hide input from a keystroke logger is the *virtual keyboard* (a.k.a. *graphical* or *on-screen keyboard*, Figure 3.1). A virtual keyboard is accessed with a pointing device, for example, a mouse or a stylus on a touch-sensitive screen. Clicking the mouse button or tapping the screen simulates a keystroke, while the position of the pointer determines the key code. On computers with regular keyboards, a virtual keyboard is used as an accessibility measure. On small handheld devices an on-screen keyboard is often the only way of typing text since these devices have limited space to include a physical keyboard. Such virtual keyboards included in the OS aim at replacing the physical keyboard at the lowest level, and therefore are not suited to protect input from keystroke logging. However, when a virtual keyboard is integrated into the application, conventional keylogging

methods will only record a series of mouse click events, while the actual keycodes remain hidden.
Integrating virtual keyboards in web applications is fairly simple. Modern browsers come with all the features required to implement them. Cascading Style Sheets (CSS) facilitate the exact layouting and positioning of HTML elements to build the user interface. Scripting languages, most prominently JavaScript, provide the application logic. The general structure of all encountered solutions is similar. The user interface consists of a single image or a group of images arranged in a grid, representing the keys. Additionally, there is a "cancel" or "backspace" button and a "submit" button, also realized with images. A handler for mouse events (e.g., `onClick`, `onMouseDown` or `onMouseOver`) is responsible for decoding and storing the input. Then the (encoded) input is submitted to the server, often with additional information. Some examples of deployed virtual keyboard solutions can be seen in Figure 3.2. They have been gathered from login pages of online-banking sites (we do not posses accounts with these banks and therefore cannot investigate beyond the login page) and attempt to protect the users password/PIN from disclosure. We have analyzed these examples to assess how well they can protect the user input from trojan-style attacks as outlined in Section 2.2.
The weakest solution is from La Banque Postale (Figure 3.2(d)), as it offers absolutely no protection from malicious BHOs, although it uses a sophisticated input method. The keys are arranged in two blocks, each block resembling the number five on a dice, with the digits 0–4 in the left block and 5–9 in the right. Within the block, the digits are randomly placed on each instantiation. To input a digit, the mouse must not be clicked, but hovered above the digit for about 0.5 seconds. This is obviously an attempt to battle screen-grabbing Trojans that attempt to grab the area around the mouse pointer when a button is clicked. Each digit is a separate image and the digit's value is encoded within the `id` attribute of the `<img>` tag. The JavaScript function triggered by the hovering extracts the value and appends it to an array. When clicking the submit button, a hidden `<input>` element is generated, and the digits from the array are concatenated to a string and set as the element's value. The resulting `HTTP POST` request contains (assuming the user has entered 1234567890 as user name and 12345 as password) the following data: `username=1234567890&password=12345`, which can be easily extracted by a BHO.
Caja Murcia (Figure 3.2(e)) has an equally poor solution which, in addition, contains a fatal bug. The PIN input field is a 5 by 2 grid of images representing the digits. The `onClick` handler of each digit calls a JavaScript function that has the digit's value as parameter and appends it to the value of a (visible) `<input type=''password''>` element. On submission, the PIN

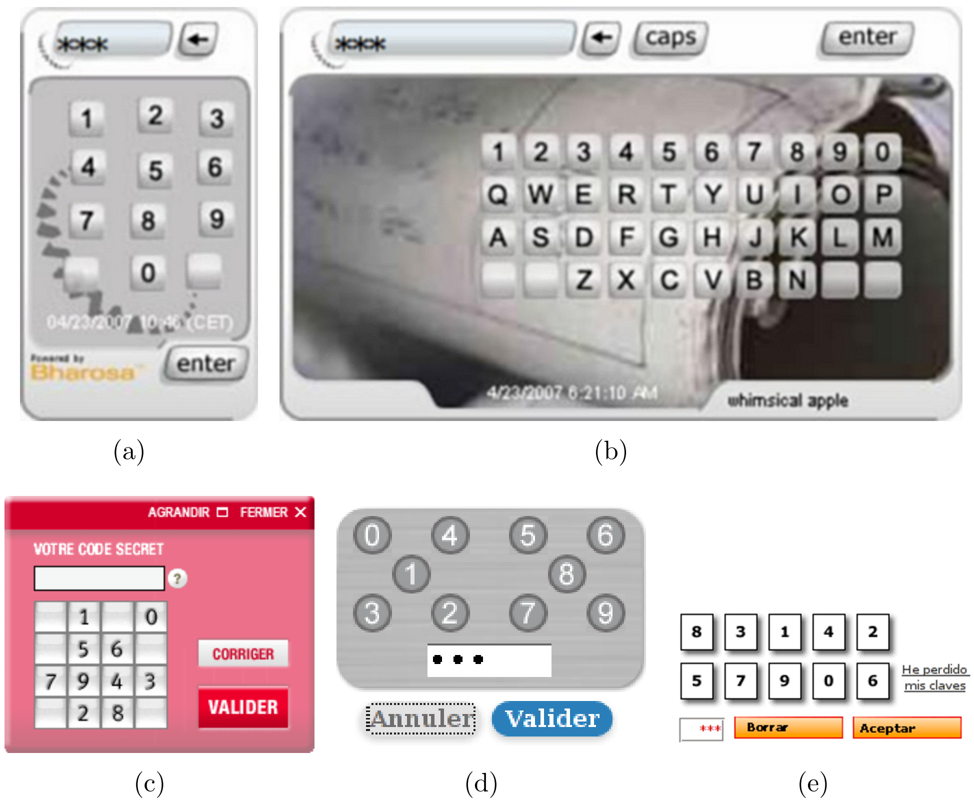


Figure 3.2: Examples of virtual keypads from online-banking portals. (a) Air Force Federal Credit Union (by Bharosa), (b) Desert Schools Federal Credit Union (by Bharosa), (c) Société Générale Bank, (d) La Banque Postale, (e) Caja Murcia

is encoded with the DES algorithm, using a Unix timestamp from a hidden frame as the key. Originally, the encoded PIN was sent *together* with the key to the server. With the ciphertext, the key and the (publicly available) DES algorithm, the PIN can be extracted easily. This was discovered and published last year [43] and while the article states that the bank failed to respond to their communication attempts, this vulnerability seems to be resolved now (i.e., the key is no longer sent with the message). However, since the key has to be transmitted *from* the server at some point in time, it is still possible, albeit more complicated, for a malicious BHO to obtain it. Tragically, all the encryption efforts are negated by a severe bug. The unencrypted PIN is stored in an `<input>` element and this element is part of the same HTML form that holds the encrypted PIN field. The plaintext PIN field is not cleared before the submission (most probably an oversight) and submitted together with the encrypted version. There, it can be grabbed as

easily as in the previous example.
A better solution comes from Bharosa, and is deployed, among others, in several US financial institutions. Bharosa offers, according to their website, "proactive, real-time fraud detection and multifactor authentication solutions for the enterprise." Bharosa virtual keyboards come in two designs, either as numeric keypad (Figure 3.2(a)) or as alphanumeric keyboard with support for upper and lower case letters (Figure 3.2(b)). In either case, the keypad consists of a single large image. When clicking on it, a JavaScript function stores the mouse coordinates (relative to the upper left corner of the image) in a string. For example, after clicking the digits 1,2,3,4,5 in the alphanumeric keypad the string looks like this: `''97,46:118,49:141,50:164,50:187,49:''`. Coordinates of clicks that do not hit digits or letters are also recorded. When submitting the input, the `HTTP POST` request contains the URL-encoded version of this string. A plain request-sniffing BHO cannot extract the password from that. However, the solution has a weakness which can be exploited by an adapted BHO. The layout and size of the keypad and the coordinate origin remain fixed — to decode the input, it is possible to hard-code the corners of each letter and digit into the malicious program and compare them with the coordinate pairs from the request.
The best solution, the only one that is potentially able to defeat a BHO Trojan, was found on the Société Générale Bank website (Figure 3.2(c)). Upon initialization, a XML file with the keypad description is fetched. The description contains the keypad dimensions (kp_{rows}, kp_{cols}), a long session ID (637 characters) and the "encoding array." In the next step, the keypad image is retrieved using the session ID. The keypad is a single image, containing a regular $kp_{rows} * kp_{cols}$ button grid where the digits 0–9 are randomly distributed (differently for each invocation) and the remaining fields are blank. To determine which digit the user wants to enter, each "button" has an associated HTML anchor element that is positioned and sized using CSS to exactly cover the button's area. Each button has an associated index value, that denotes the button's position inside the grid by simply numbering them $0..(kp_{rows} * kp_{cols} - 1)$ from upper left to lower right. Therefore there is no connection between the button's index and value. The `onClick` handler of the anchor element encodes the button index and the position of the entered digit inside input as a single number. It calculates an index $((kp_{rows} * kp_{cols}) * pos_{digit} + index_{button})$ for the encoding array. The value retrieved from that array is appended to a code string. After entering 123456 the code string might look like this: `75,136,88,164,134,39`. Upon submission this code string is sent to the server together with the session ID. Even though the encoding scheme can be reversed when the array content is

known, simple request-snooping will not reveal the input, because the true values of the entered digits do not appear at any point inside the application. The random placement of the digits in the input image prevents attacks that are effective against Bharosa solutions. To decode the actual input, the image has to be analyzed.
The virtual keyboard solutions have been in use for some time now, and while some do not raise the difficulty for the attacker at all, others show formidable resistance against traditional methods. Malware authors start to adapt to these new challenges by including screen-capturing facilities [20, 43]. The best method to defeat virtual keyboards is to make a screenshot (including the cursor) each time a mouse button is clicked. Most operating systems have a built-in screenshot utility which can be abused for that purpose but that is beyond the scope of a BHOs capabilities. The malicious program would have to be anchored at a lower level of the system. To extract any useful information (e.g., a PIN) from a series of captured images the attackers need to analyze them. The simple approach is to analyze them manually, that is, to have a human look at them. The delay imposed by that does not mitigate the threat to "permanent" security tokens like passwords or PINs, which are likely to remain unchanged for an extended period of time. To automate the extraction (and make it instantaneous), the malware might have built-in Optical Character Recognition (OCR) or other image analysis features (possibly tailored to overcome a specific protection scheme). So far, there have been no reports of such malware, but with the raising popularity of on-screen keyboard based input protection and many freely available OCR solutions, it seems like the next logical step.

3.3 Confirmation Methods

We have to concede, that every existing scheme protecting the *confidentiality* of sensitive user data (security tokens and the data they guard) can fail, at the latest with the intervention of a human attacker. To mitigate the damage imposed by stolen user credentials, most banks[2] use a second layer of protection to safeguard the *integrity* of their customer's finances. To issue any kind of financial transaction, the user needs to confirm his identity (again) by providing a security token that is either single-use (or valid for a limited time only) or bound to a physical item the users possesses. Currently the following methods are in use and some fit both categories:

[2]This statement is true for most European banks. The two-level authentication scheme is not very common in U.S. banking institutions

- Single-use:
  - TAN
  - Indexed TAN (i-TAN)
  - TAN-Generator (sm@rt-TAN)
  - Mobile TAN (m-TAN)
- Physical:
  - Smart Card + TAN-Generator (sm@rt-TAN)
  - Smart Card with Digital Signature Function + Card Reader
  - Digital Signature on (Floppy, Compact) Disc (deprecated)

Transaction Authentication Numbers (TANs) are short (4–8 digits) randomly generated numbers. The distinguishing feature of the many TAN incarnations is the method how these numbers can be obtained. In the simplest form, the customer receives a list of TANs in a hardcopy letter via regular post delivery services from her banking institute (the bank uses the regular post delivery service as a second secure communication channel to the customer). A typical list has 50 TANs, what is considered to be sufficient for 6 moths of regular use. To complete a transaction, the customer has to pick a random TAN from the list and enter it. Each TAN in the list can be used once, and when a certain amount of TANs has been used up, the customer receives a new list. It is general practice that only the two most recent lists are valid; all unused TANs on the older list become invalidated when a new one is received. Since a TAN is only entered once to confirm a transaction and is invalidated upon completion, simply stealing it is of no benefit to the attackers. To obtain a valid TAN with a Trojan attack, the malware has to capture the user input[3] and intercept the submission of the transaction form. The captured data is transmitted to the attackers while a fake confirmation message is displayed to mislead the user. Then the attackers are free to use the still valid TAN.
A different scheme to obtain valid TANs (or personal data in general) is called "phishing." A phisher tries to lure (hence the idiom) the victim into handing over sensitive information, by pretending to be a trustworthy entity. This is a general social engineering technique that predates the web (and computers) by far, and "phishing" is only the name for it's extension into the Internet domain. We mention phishing only for the sake of completeness,

[3]A simple keylogger is sufficient. To our knowledge, no online-banking portal protects the input of TANs with virtual keypads yet

since it is a current and related problem, but further elaboration would go beyond the scope of this thesis.
To counter the TAN-stealing Trojans (and phishing attacks) several enhanced versions of TAN have been introduced over the years. The Indexed TAN (i-TAN) is requested by the web application instead of picked randomly by the user. More specifically, the TANs in the list are numbered serially and the application asks for a TAN with a specific number, for example, "Please enter the TAN number that starts with 48." Only this TAN can be used to confirm the transaction, all others are invalid at this point in time. A TAN-stealing Trojan can obtain only one TAN at a time, and with a list of 50 TANs to choose from, the chance that this TAN will be requested when the attackers attempt to make a transaction is 2%. While phishers have adapted to this situation (they now request 10 or 20 TANs at once), attackers using Trojans would need multiple attempts, at which point even the most uninformed users should turn suspicious.

(a) Simple TAN Generator

(b) Generic Smart Card

(c) Smart Card Reader

Figure 3.3: Examples of hardware-based confirmation solutions.

All further methods to improve TAN security resort to external hardware. TAN generators (Figure 3.3(a)) are small handheld devices that will generate a series of valid TANs, one at a time. They can work stand-alone, or in tandem with a smart card. In both cases, a shared secret between the bank and the customer must exist, that will produce, through the generation algorithm, the same series of tokens on both sides. However, this method seems only to mitigate the (rather insignificant) threat of intercepting the TAN list in the mail since it is vulnerable to the steal-and-deceive trojan-based attack. The mobile TAN (m-TAN) idea was inspired by the ubiquity of mobile phones and the popularity of text messaging services. It is held to be the most secure TAN-based scheme so far. A transaction consists of three steps. First, the user submits the transaction details to the server. Then, the server arranges for a text message to be sent to the users mobile phone, containing the TAN. The TAN is valid for limited time only and bound to the submitted data. Finally, the user submits this TAN to conclude the transaction. This thwarts steal-and-deceive Trojans, since the obtained TAN is of no use for a different transaction, and phishing attempts, since TANs are only issued when required. However, all the TAN-based methods are still vulnerable to the request-tampering BHO presented in Section 2.2.

An alternative to TAN-based solutions is the digital signature [30] method. Digital signatures are based on public-key cryptography and used to prove the integrity of the message and the identity of the sender at the same time. In many countries, digital signatures are legally equivalent to handwritten signatures. A digital signature is generated by encrypting a checksum of the signed message with the sender's private key. The signature is then sent with the message and the recipient decrypts the signature with the public key of the sender. The resulting number is compared with the checksum which is computed anew from the received message — if they match the message is authentic and has not been modified. In web banking applications the "message" is the transaction data (account, amount, etc.) and the private key is stored on a smart card (Figure 3.3(b)). Storing the key on a floppy or compact discs is also supported by some (legacy) systems, but that method has serious drawbacks and it's use is discouraged. Since data discs can only *store* information, an application (on the potentially compromised computer) must retrieve the key to perform the encryption and exposes it hereby to the risk of theft. In contrast, (cryptographic) smart cards never reveal the key and perform the encryption on-chip.

To use a smart card at home, additional hardware (a smart card reader, Figure 3.3(c)) is required. A smart-card-confirmed transaction works as follows: The transaction data is submitted to the server via a regular web form. Then the data is sent back from the server and displayed in a "secure viewer" (e.g., a Java applet). The secure viewer sends the data via a trusted path to the card reader. After the user enters the card PIN (using the reader's keypad), the smart card signs the data and sends the signature back to the viewer. Finally, the viewer submits the data together with the signature back to the server. None of the attacks previously introduced here are able to break this

scheme, but new attacks targeting the smart card infrastructure, for instance, the card reader's driver [27, 48] or the secure viewing applet[4], have been proposed. For the customer, the smart card solution indeed brings heightened security, but at the penalty of additional hardware (and the associated software and drivers) that restricts the mobility and might not work on all platforms. This limits the "anytime, anywhere" aspect of web applications.
In a recent development (December 2006), a novel TAN generation method has appeared on the market. To date, and to our knowledge, only one online-banking site [3] uses it. The method is, in some aspects, similar to one of the ideas presented in this thesis (Section 5.2). However, work on this thesis has started three months prior to that publication and the ideas presented here do not rely on additional hardware. The new TAN generator has a numeric keypad, where the destination account of the transaction is entered, and the TAN is computed from this data and a timestamp, binding the TAN to the input. There is little additional information on this method, therefore we cannot make any assumptions on it's efficacy.

[4]Research on that topic has been carried out at our department just recently, but unfortunately is covered by non-disclosure agreements

Chapter 4

CAPTCHA

In this thesis, we present a new method to protect user input in web applications from automated exploitation techniques introduced in Chapter 2. We attempt to create a graphical keyboard that is resistant to the attacks outlined in previous chapters by combining the virtual keyboard idea with a CAPTCHA challenge. Virtual keyboards have been covered extensively in Section 3.2 and this chapter will explain the origins and the current applications of CAPTCHAs. The name "CAPTCHA" is an acronym of "Completely Automated Public Turing test to tell Computers and Humans Apart." It is a challenge-response mechanism that potentially allows a computer to determine if the challenged party is a human or a machine and is a variation of the test to determine machine intelligence first proposed by Alan Turing in 1950 [49].

4.1 The Turing Test

Soon after the introduction of (electronic) computers in the 1940ies, people began to speculate if such machines could ever attain human-like intelligence. The mathematician Alan Turing, in his article "Computing machinery and intelligence" [49], stated, that the question "Can machines think?" is too ambiguous. Instead, he proposed a test based on the "imitation game," a then popular parlor game. In the original game, a judge would converse (without personal contact) with a man and a woman, and while the man would try to convince the judge he is the woman, she would attempt to affirm her true identity. The task of the judge is to correctly assess who is who. In Turing's version, the judge and the truthful contestant would be human, and the machine would be the deceiving party. When the judge, based on the answers he receives from both contestants, is unable to determine which one is the machine, then the machine “passes” the test. Therefore, the new question posed by Turing is: Can a machine make a human interrogator believe it is a human, when the interrogator has to base his decision solely on the machines “mental” capabilities (excluding any physical characteristics)? Turing believed, that it is only a matter of storage capacity and processing power, and that by the end of the 20^{th} century computers would be powerful enough to pass the test. This turned out to be overly optimistic, even with a rise of storage and performance way beyond Turing’s expectations. Additionally, over the years, criticism has arisen if the Turing Test[1] can truly measure machine intelligence, or just if a machine is good at imitating a human (and if there is a difference between these two). Nevertheless, the challenge to build a computer that would pass the test inspired many advances in artificial intelligence (AI) and machine learning. One of the more recent “byproducts” is the CAPTCHA.

4.2 CAPTCHA - An Automated Turing Test

The idea of using a Turing Test to verify that a human is making a request to a web service can be traced back to an unpublished manuscript in 1997 [37]. The first practical implementation was deployed by AltaVista, a search engine and free web hosting provider, around the year 1998 [28] to prevent automated “bots” from registering for web services. Academic work on this topic started soon thereafter at the Carnegie Melon University (CMU) [8, 50, 51], and there the term CAPTCHA was coined (and trademarked). The key difference between a regular Turing Test and the Automated version is that the judge is now a machine. The term “Reverse Turing Test” is sometimes also used but is considered inaccurate and ambiguous (other possible reversions exist, e.g., both contestants might try to convince the human judge that they are a machine). The task of the judge remains the same: Pose challenges to the contestants to determine which one is human. Since the test is supposed to bar machines from accessing a web service while simultaneously allowing humans to pass, the challenges must be devised in such a way that a machine will fail with a high probability (optimally 100%) while a human will pass most of the time. At first glance, this seems paradox, since the judge is also a machine, and therefore unable to solve the challenge itself. However, the judging machine does not need to solve the challenge, it only needs to know the correct answer(s) and compare them with the solution submitted by the contestant. Additionally, if it can be proven (or assumed with high

[1]This name was made popular by Arthur C. Clarke in his 1968 science-fiction novel “2001: A Space Odyssey,” 14 years after Turing’s death

probability) that the challenge is impossible to solve for a machine, there is no need for a parallel interrogation like in the original test and a single challenge-response cycle is sufficient (instead of a series) to positively decide the outcome. Finally, the "P" in CAPTCHA stands for "public," meaning that the hardness of the challenge shall not rely on obscurity of the algorithms involved, but rather the problem itself is "hard" in a mathematical sense. This is analogous to public-key cryptography, where the encryption algorithms are publicly available and the effectiveness is based on the assumption (and backed with mathematical proofs) that with state-of-the-art technology, it is impossible to extract or guess the decryption key in reasonable time.
In case of the CAPTCHA, which was born from a test of machine intelligence, the challenge is based on an (hard) AI problem. Or, in the words of the CAPTCHA inventors from CMU [50]:

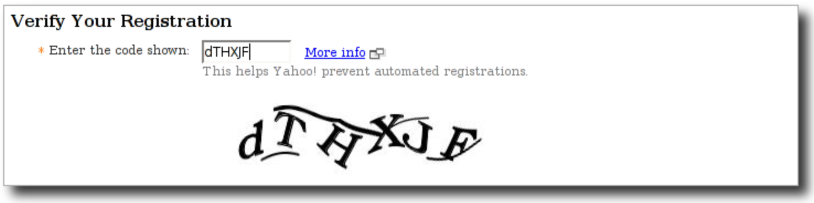
> *A* CAPTCHA *is a cryptographic protocol whose underlying hardness assumption is based on an AI problem.*

Since Turing's times, AI research has diverged from the goal to create a human-like machine intelligence, and instead focuses on specialized solutions for different AI problem classes (e.g., chess, expert systems, machine vision, natural language). To create a good CAPTCHA, one need to consider the classes of problems where humans (by far) exceed the best AI solutions. Such problems, for example, are the understanding of natural language or the recognition of visual patterns. In fact, most of the CAPTCHAs in use are based on these problems.
The visual CAPTCHA consists of an image or a set of images. The challenge for the contestant is to identify the subject matter of these image(s) and base the answer on that knowledge. Several kinds of visual challenges are in use. Text recognition CAPTCHAs are prevalent, but there are also some object recognition or object classification challenges. Novel visual challenges are a topic of ongoing research (see Section 4.3).
A natural language CAPTCHA could have the form of a "common sense" question, for instance, "What color is the sky?" To be effective, this would require a large repertoire of questions. Another option is posing mathematical tasks in textual form, for example, "What is the sum of twenty-three and forty-two?" However, there is only a limited amount of possible ways to pose such a question and the vocabulary is strictly defined, so building a parser able to "understand" the problem is imaginable. A third possibility is to use analogy questions known from aptitude tests, for instance, "car : driver = airplane : ???" with several choices presented for the missing word. With a large enough database of analogies, this might also be effective.

A different kind of language CAPTCHA is the aural CAPTCHA. Aural CAPTCHAs were introduced after the lack of accessibility in visual CAPTCHAs was criticised. To solve the challenge, the user has to understand what a voice in a (slightly distorted) audio recording is saying. Although language (and aural) CAPTCHAs have certain advantages with regard to accessibility (see Section 4.5), they are not widely used. The prevalent type of CAPTCHA in use is the visual CAPTCHA, mainly because it's easy to implement (and the web is a visual medium after all).

4.3 Visual CAPTCHAs

The oldest (and still most widely used) kind of CAPTCHA challenge is based on the recognition of text [28]. A word (or a series of meaningless characters) is printed in the image. To positively solve the challenge, the user must identify the characters in the image and enter them in a text field (Fig. 4.1). The hardness assumption is that it is not possible for a machine to identify the characters using Optical Character Recognition (OCR). OCR algorithms are designed to translate images of handwritten or typewritten text into machine-editable text. That is, it is not enough to display simple text to users graphically as a form of CAPTCHA. Such graphics can be analyzed using OCR and the embedded text can be easily extracted. To defeat OCR, CAPTCHAs generally use background clutter (e.g., thin lines, colors, etc.), a large range of fonts, and font transformations. Such properties have been shown to make OCR analysis difficult. In contrast, the human brain and visual system is very good at recognizing patterns (e.g., letters) in incomplete, obscured and/or vague information.


Verify Your Registration

* Enter the code shown: dTHXJF More info

This helps Yahoo! prevent automated registrations.

dTHXJF

Figure 4.1: Part of the Yahoo! registration page including the CAPTCHA challenge.

A different class of CAPTCHA is based on object recognition [15, 17]. The user is shown an image/set of images and has to identify the object contained in it/them (Fig. 4.2). This is called a *naming* CAPTCHA. A variation of this theme is the *distinguishing* CAPTCHA, with two sets of images. Each set contains pictures of the same subject. The challenge is to identify if both sets show the same subject or not. Another possibility is the *anomaly* CAPTCHA, where the user has to pick, from several images, the one whose subject differs from all the others. Or, there are several different objects in a single picture and the user has to correctly name some of them. As with text-based CAPTCHAs, distortions of the image can be used to make the recognition task harder for machines. To create effective object-recognition CAPTCHAs, a sufficiently large database of (correctly) annotated images is required. Additionally, potential ambiguities (e.g., an object having many synonyms or a word having multiple definitions) have to be avoided.

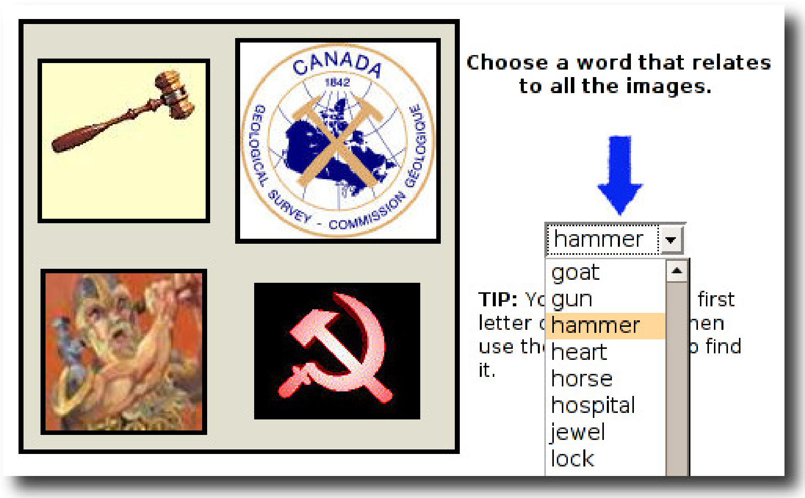
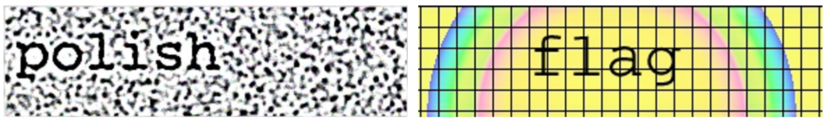


Figure 4.2: PIX - An object naming CAPTCHA.

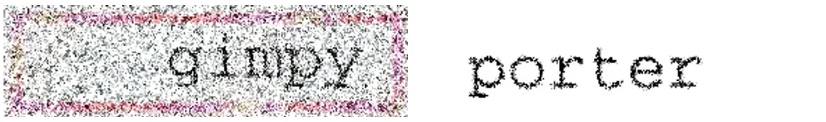
4.4 Breaking Visual (Text Recognition) CAPTCHAs

The recognition of printed text to convert it in a digital form (e.g., Unicode character codes) that can be processed by computers is an active research topic for as long as electronic computers exist [34]. The recognition of printed Latin characters, regardless of the typeface used, is considered a solved problem as of today, Optical Character Recognition (OCR) algorithms for Latin exceed 99% accuracy. Hand-printed or cursive script and alphabets with a large number of characters (e.g., traditional Chinese) produce much lower success rates and are still subject of active research. The designers of the earliest CAPTCHA implementations [28] looked at state-of-the-art

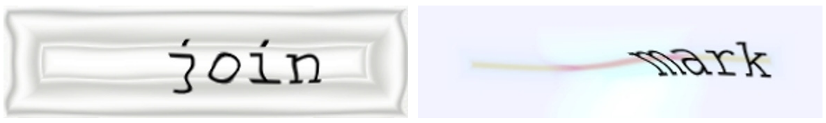
OCR solutions and tried to construct cases which, according to the OCR software manuals, would produce poor results. This included background clutter (Fig. 4.3(a)), characters with fuzzy or blurry edges (Fig. 4.3(b)) and warped text (Fig. 4.3(c)) — effects that often naturally occurred when documents were photocopied or faxed.



(a) Background clutter



(b) Blurry characters



(c) Warped text

Figure 4.3: Examples of early CAPTCHAs (EZ-Gimpy).

Soon, researchers and bot creators alike started to devise ways of “breaking” (i.e., machine-solving) these CAPTCHAs. The EZ-Gimpy CAPTCHA (Fig. 4.3) has been broken with a success rate of 92% by Mori et al. in 2003 [33]. This is partially due to the fact that EZ-Gimpy uses a dictionary of approximately 600 short English words (4–6 letters), so a dictionary can be used to identify the word, even if a few characters cannot be recognized with high confidence. Other CAPTCHA generation algorithms have appeared, with varying degrees of sophistication. They were built on the same principles as EZ-Gimpy, but at least used combinations of random characters (upper/lower case letters, digits, non-alphanumeric) instead of dictionary words. Still, many of them are considered to be broken (Fig. 4.4). Unfortunately, most of the breaking attempts did not occur in an academic setting and are poorly documented, which hardly makes them a citeable source (e.g., the PWNtcha CAPTCHA decoder [22]). Still they help to identify common weaknesses of CAPTCHAs, for example, constant font, aligned characters, predictable character positions, no rotation and no perturbation.
Researchers have identified the two steps necessary to successfully break a textual CAPTCHA: *segmentation* and *recognition*. Segmentation is required to pin-point the position of each character in the image and identify all the pix-

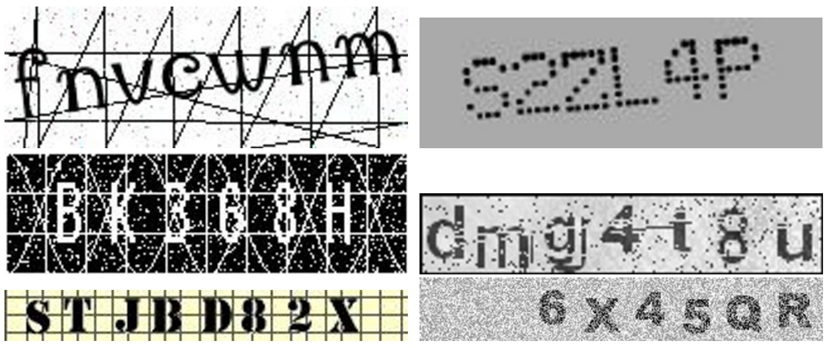
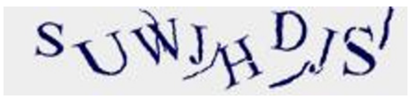


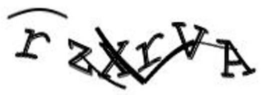
Figure 4.4: CAPTCHAs allegedly solvable by the PWNtcha CAPTCHA decoder with near 100% success rate.

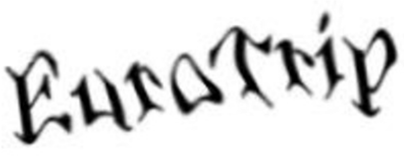
els belonging to it while removing any disturbing clutter. Then recognition is performed on each segment to determine the character it represents. Recent advances in OCR [11, 13] allow machines to perform en par with or even surpass humans in in some cases of (printed) character recognition. To overcome this, CAPTCHAs based on hand-written text [40] have been proposed. Recognition of handwriting is still considered a hard problem and existing solutions (e.g., post address interpretation) require contextual knowledge and a constrained dictionary to perform adequately.
Other CAPTCHA algorithms tend to focus more on the segmentation task. In the human visual system (HVS), segmentation is an important preprocessing step that occurs partly in the retina and partly in different regions of the brain (lateral geniculate nucleus and the visual cortex). Many properties of the HVS are still not well understood and it is difficult to create algorithms that would emulate them. The segmentation-based CAPTCHAs try to exploit such inherent properties of the HVS. For example, humans are able to "see" complete shapes even when some parts are missing or some superfluous parts are added. Most researchers agree, that segmentation is a harder (AI) problem than character recognition, which is reflected by the new types of CAPTCHAs developed by them.
ScatterType [4] tries to make segmentation for machines very hard by splitting the characters vertically and horizontally in many parts and drifting them apart. The Microsoft Passport CAPTCHA [10] (Fig. 4.5(a)) uses a combination of non-affine text transformation (global and local warp) and intersecting geometric shapes in foreground (hinders segmentation and creates false positives) and background (breaks characters apart) colors. Another recently proposed CAPTCHA [19] exploits the masking[2] property of the HVS.

[2]Masking is the effect when interference between two different visual signals either

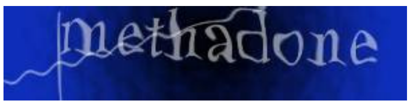
Through the balanced application of textures, edges and noise these CAPTCHAs have an interesting property: Some of the characters can be identified by computer vision while being (deliberately) invisible to humans.
While ScatterType and the masking CAPTCHA are, to date, only academic propositions, Passport and some similar CAPTCHAs (Fig. 4.5) have been deployed on popular websites for some time now, and no reports of successful breaking have surfaced yet. This gives confidence that it is indeed possible to develop effective CAPTCHAs which are not machine-solvable with very high probability. If, in the future, algorithms are developed to solve these CAPTCHAs, it is conceivable that even better generation algorithms can be developed as well.


(a) Passport


(b) Yahoo!



(c) IMDb



(d) CFXCaptcha

Figure 4.5: CAPTCHAs currently deployed on popular websites with no reports of successful breaking.

4.5 Criticism on Visual CAPTCHAs

Although CAPTCHAs have proven effective against automated web service abuse, they have met with criticism. Some people pointed to the successfully broken CAPTCHAs to question the efficacy of the approach in general. This is partially due to the many "homebrewn" CAPTCHA implementations based on the AltaVista CAPTCHA and EZ-Gimpy, which indeed have been compromised. However, the new generation of CAPTCHAs, based on sound research of OCR algorithms and human perception, demonstrates great resistance to breaking attempts.
A more justified criticism comes from web accessibility advocates [52]. Since visual CAPTCHAs are by design not machine interpretable, common assis-

increases or decreases their visibility.

tive technologies like screen readers and braille displays which transform the on-screen text in alternative forms cannot be used. This effectively bans people with disabilities, who are dependant on these technologies to surf the web, from accessing a CAPTCHA protected service. Matt May from the W3 Consortium states it this way:

> *In many cases, these systems make it impossible for users with certain disabilities to create accounts, write comments, or make purchases on these sites, that is,* CAPTCHA*s fail to properly recognize users with disabilities as human.*

Although the W3C proposes some more accessible alternatives, it admits that none are entirely satisfactory and some (e.g., universal biometric identification) are not realizable with today's deployed technology. Indeed, it seems that web security, which inherently tries to restrict access, and web accessibility are somehow contradictory, unless completely new paradigms for web security *and* accessibility are developed.
A further point of criticism is targeted at the hardness of some (newer) CAPTCHAs, which are sometimes reported to be unsolvable even for humans without disabilities. While the new visual CAPTCHAs like Passport (Fig. 4.5(a)) or Yahoo (Fig. 4.5(b)) are very effective against OCR attacks, some unfortunate configurations might be unreadable for humans, too. There is active research on the topic of how hard a CAPTCHA can get before humans start to fail it in numbers [12].

Chapter 5

Proposed Solutions

As described in Chapter 2, the web application must assume that the user's web client (and platform) is under the control of an attacker. There are two aspects of the communication that an attacker could compromise: the confidentiality, and the integrity of input sent from the client to the web application. The confidentiality of the input is compromised when the attacker is able to eavesdrop on the entered input and intercept sensitive information. Analogously, the integrity of the input is compromised when the attacker is able to tamper, modify or cancel the input the user has entered.
As far as the user is considered, there are cases in which the integrity of input may be more important than its confidentiality. For example, if the attacker is able to read the account number that the victim has provided to an online banking web application (i.e., the confidentiality of the entered account number is compromised), this information cannot be used by the attacker to perform an illegitimate money transaction. However, if the attacker can *modify* the account number that has been typed as described in the example in Section 2.2 (i.e., the integrity of the entered account number is compromised), an illegitimate money transaction can be performed, causing a financial damage to the user and/or the bank.
In this chapter, we present two techniques that web applications can apply to protect sensitive user input. We assume a threat model in which the attacker has compromised a machine and installed malicious code. This code has complete control of the client's machine, but must perform its task in an autonomous fashion (i.e., without being able to consult a human). This assumption is realistic, since an attacker will typically infect a large number of machines (to maximize the the effort-benefit ratio) and will be unable to monitor all the infected machines in real time.
Our solutions are implemented on the server and are client-independent. They are primarily aimed at protecting the *integrity* of financial transac-

tions. Specifically, we present measures that should prevent any successful tampering of the transaction details to redirect user-initiated transactions to destinations of the attacker's choosing. Additionally, we design the confirmation tokens in such a way, that stealing them does not enable the attacker to use them for a different transaction. The secure input method we present has the additional bonus of protecting the *confidentiality* of the transaction, at least from automated attacks. Unfortunately, as we had to concede in Section 3.3, total protection of confidentiality is not possible since even if the Trojan cannot analyze all data itself, it can always pass it on to a human attacker for later (off-line) analysis.

5.1 Solution 1: Using CAPTCHAs for Secure Input

5.1.1 Overview

The basic idea of the first solution is to extend graphical input with CAPTCHAs [8]. We have covered the different types of CAPTCHA in Chapter 4. To recap, a CAPTCHA is a test that:

- a machine can generate and grade
- a human can easily pass
- a machine is unable to pass

A graphical input method that adheres to these principles would be safe from tampering by any automated attack (e.g., from the Trojan attacks presented in Chapter 2).

5.1.2 Description

Although CAPTCHAs are frequently used to protect online services against automated access, to the best of our knowledge, no one has considered their use to enable secure input to web applications. On the other hand, graphical keyboards are already deployed on several online-banking sites to protect identification tokens such as passwords or PINs. While several poor implementations can be bypassed by traditional means, even the best solutions are susceptible to OCR attacks. Moreover, we did not encounter graphical keyboards used to protect user input from tampering, for example, when entering the destination account for a money transfer.

In our solution, whenever a web application requires to protect the integrity of user information, it generates a graphical input field. Then, CAPTCHA characters are randomly placed on this field. When the user wants to transmit input, she simply uses the mouse to click on the area that corresponds to the first character that should be sent. Clicking on the image generates a web request that is then sent to the web application. This request contains the *coordinates on the image* where the user has clicked with the mouse. The key idea here is that *only* the web application knows which character is located at these coordinates. After the first character is transmitted, the web application generates another image with a different placement of the characters, and the process is repeated. By using CAPTCHAs to communicate with the human user, a web application can mitigate client-side attacks that intercept, or modify the sensitive information that users type. Because the CAPTCHA characters cannot be identified automatically, a malware program has no way to know which information was selected by the user, nor does it have a way to meaningful select characters of its own choosing.

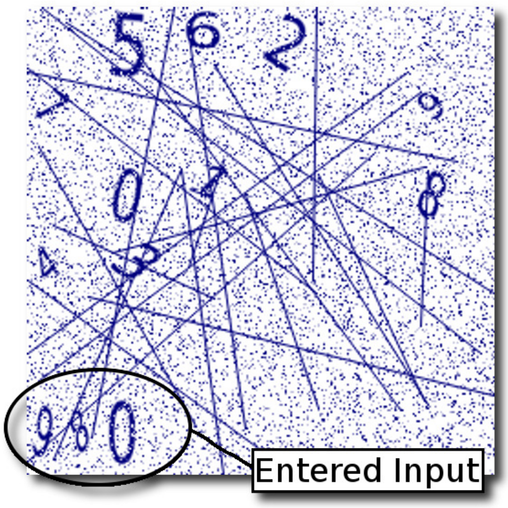


Figure 5.1: A generated CAPTCHA for enabling secure user input. Note that the input the user has entered, 980, is also displayed as a CAPTCHA.

Figure 5.1 depicts a screenshot of our prototype CAPTCHA input element. In this example, the user would like to securely enter her bank account number 980.243.276. It can be seen that the web application has generated digits from 0 to 9 that are randomly distributed on the upper portion of the image. To transmit the account information, the user locates the desired numbers on each generated image and sequentially enters the account number that she would like to send. In order to give immediate feedback to the user about the number she has just entered, a part of the CAPTCHA can be used to reflect back the input. In our prototype implementation, we chose to use the bottom line of the CAPTCHA to give feedback to the user about the number she has just entered (see the marking in Figure 5.1; the first three digits have already been entered). Of course, one can also allow the user to modify the input she has entered by including input elements in the CAPTCHA that can be used for deleting and editing.

5.1.3 Possible Attacks

The security of the CAPTCHA-based solution is based on the assumption that it is impossible (or sufficiently difficult) for malicious software installed on the user's machine to analyze and identify the coordinates on the CAPTCHA where the input values are located. Our solution also relies on the presumption that we can adapt and integrate recent advances in CAPTCHA research and hence, match advances in OCR research.

In order to successfully defeat our solution automatically, the malicious software would need to identify the mouse coordinates that it needs to send to the web application. Suppose, for example, that a Trojan horse would like to send the account number 276.173.862 to the bank instead of the original account number 980.243.276 that the user would like to enter. Each time a CAPTCHA input element is generated and displayed by the web application, it would have to automatically identify the target input value that it requires, intercept the user's mouse click and simulate a mouse click of its own. In our example, the Trojan would first have to identify the number 2 in the first generated CAPTCHA, the number 7 on the second generated CAPTCHA, and so on. Since the input is reflected in the CAPTCHA, the user is bound to notice that instead of 980 that she entered, 276 is displayed — this is another obstacle for the attacking Trojan. To successfully deceive the user, the Trojan would need to retouch the image (remove the original digits from the image and replace them with the ones the user entered, while keeping the appearance of the image intact). A human with image-editing software like Photoshop would need a few minutes to do that and a machine performing such a task is, to date, unheard of.

Although algorithms exist that can make it sufficiently difficult for software to solve CAPTCHAs automatically, the main problem currently is that it may be possible to relay CAPTCHAs to human operators that can then solve them manually. In fact, some spammers have been using this technique in real-life to defeat CAPTCHAs [52]. One popular idea is to divert the CAPTCHA to another web site where visitors willingly solve them because they believe that they will access "free" pornographic content. Note, however, that in the solution we propose, it is more difficult to deploy an unsuspecting human operator to defeat the CAPTCHA. That is, in our case, the required task is not simply reading the content of the CAPTCHA, but identifying input values sequentially. Hence, a human operator would have to follow the input that is being entered in real-time while the victim is entering this information during an online banking session. Additionally, he would need to retouch each image before it reached the user.

The attackers could nevertheless deploy human operators that indeed analyze and enter information on CAPTCHA-based input elements. One can imagine, for example, that the miscreants would employ and pay these operators for their services. However, in this case, the attackers would have to make sure that the communication between the Trojan horses on the victims' machines and their human operators are reliable and fast. Furthermore, they would also run the risk of being traced back if service providers monitor network connections and detected Trojan horses are analyzed for their behavior. Thus, we believe that the CAPTCHA-based input solution we propose considerably raises the difficulty bar for attackers and significantly increases the costs of client-side attacks.

5.2 Solution 2: Binding Sensitive Information to Confirmation Tokens

5.2.1 Overview

The second solution is to based on *confirmation tokens*. In principle, the concept of a confirmation token is similar to a transaction number (i.e., TANs) commonly used in online banking (see Section 3.3).

TAN-based schemes rely on the assumption that an attacker will not have access to a user's TAN list and hence, be able to perform illegitimate financial transactions at a time of his choosing. However, TAN-based schemes are easily defeated if the attackers perform a client-side attack (e.g., using a Trojan horse as described in Section 2.2). Furthermore, such schemes are also vulnerable to phishing attempts in which victims are persuaded to give away their TAN numbers that are then used by the attackers to perform illegitimate financial transactions.

The problem with regular transactions numbers is that there is no relationship between the data that is send to the web application and the (a-priori shared) TANs. Thus, when the bank requests a certain TAN, malicious code can replace the user data without invalidating this transaction number. To mitigate this weakness and to enforce integrity of the transmitted information, we propose to bind the information that the user wants to send to our confirmation token. In other words, we propose to use confirmation tokens that (partially) depend on the user data. Note that when using confirmation tokens, our focus is not the protection of the confidentiality, but the integrity of this sensitive information.

5.2.2 Details

Imagine that an application needs to protect the integrity of some input data x. In our solution, the idea is to specify a function $f(.)$ that the user is requested to apply to the sensitive input x. The user then submits both her input data x and, as a confirmation token, $f(x)$.
Suppose that in an online banking scenario, the bank receives the account number n together with a confirmation token t from the user. The bank will then apply $f(.)$ to n and verify that $f(n) = t$. If the value x, which the user desires to submit, is the same as the input n that the bank receives ($x = n$), then the computation of $f(n)$ by the bank will equal the computation of $f(x)$ by the user. That is, $f(x) = f(n)$ holds. If, however, the user input is modified, then the bank's computation will yield $f(n) \neq f(x)$, and the bank will know that the integrity of the user's input is compromised.
One important question that needs to be answered is how $f(.)$ should be defined. Clearly, $f(.)$ has to be defined in a way so that malicious software installed on a user's machine cannot easily compute it. Otherwise, the malware could automatically compute $f(x)$ for any input x that it would like to send, and the proposed solution fails.
We propose two schemes for computing $f(x)$. The first scheme, called *token calculation*, uses a collection of simple algorithms that are sent to users in a *code book* (i.e., similar to TAN letters described in the previous section). The code book contains a sufficient number (e.g., 100) of simple algorithms that can be used by users to *manually* compute confirmation tokens (e.g., similarly to the obfuscation and challenge-response idea presented in [14] for secure logins). All algorithms are based on the input that the user would like to transmit.
Suppose that the user has entered the account number 980.243.276, but a Trojan horse has actually sent the account number 276.173.862 to the bank. The user does not suspect anything and sees 980.243.276 on his screen. In the first scheme we propose, the bank would randomly choose an algorithm from the user's code book. Clearly, in order to make the scheme more resistant to attacks, a different code book would have to be created for each user (just like different TANs are generated for different users). Figure 5.2 shows an excerpt from our sample token calculation code book. Suppose the bank

. . .

Token ID 4:
Create a number using the 5th and 7th digits of the target account and add 542 to it.

Token ID 5:
Create a number using the 3rd and 5th digits of the target account and add 262 to it.

Token ID 6:
Multiply the 4th and 8th digits of the target account and add 17 to the result.

Token ID 7:
Create a number using the 3rd, 6th and 7th digits of the target account.

. . .

Figure 5.2: Excerpt from our sample token calculation code book.

asks the user to apply algorithm ID 6 to the target account number. That is, the user would have to multiply the 4th and 8th digits of the account number and add 17 to the result. Hence, the user would type 31 as the confirmation token. The bank, however, would compute 23 and, because these confirmation values do not match, it would not execute the transaction and would thwart the attack.

For our second scheme to realize $f(.)$, called *token lookup*, users are not required to do any computations. Suppose that the user is faced with the same attack that we discussed previously. That is, the user enters 980.243.276, but the malicious application sends 276.173.862 to the bank. In this variation, the code book would consist of a large number of random tokens that are organized in *pages*. The bank and the user previously agree on *which* digits of the account number are relevant for choosing the correct page. The bank then requests the user to confirm a transaction by asking him to enter the value of a specific token (on that page). For example, suppose that the relevant account digits are 2 and 7 for user John and that the bank asks John to enter the token with the ID 20. In this case, John would determine the *relevant* code page by combining the 2nd and 7th digits of the account number (82 in our case) and look up the token on that page that has the ID 20 (see Fig. 5.3). However, because the attacker has sent another account number to the bank, the token that John transmits would not be accepted as valid.

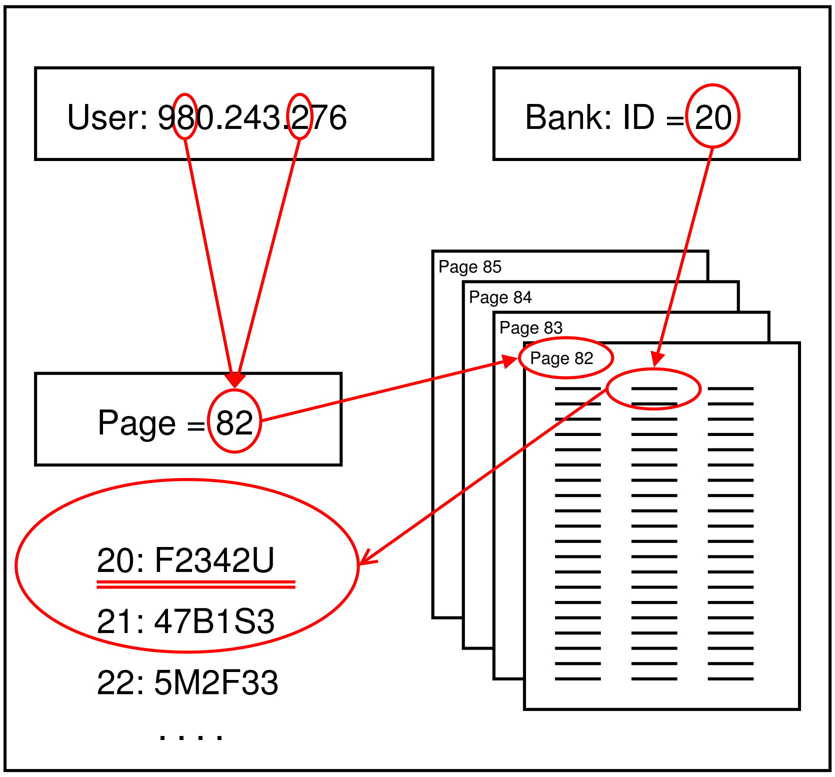


Figure 5.3: The token lookup scheme.

5.2.3 Possible Attacks

The security of the confirmation-token-based solutions relies on the assumption that the relevant information remains secret. In the case of the computation scheme, that means the algorithms the bank asks the user to apply are not revealed or can be deduced from the account number and the computed confirmation token. The lookup scheme can be compromised, when both the relevant digits for code lookups and the contents of the code book are disclosed.
Note that it is possible in theory that some digits of the target account of the attacker correspond to the digits in the target account number of the user. Hence, it could be possible that an algorithm is chosen randomly by the bank that computes the *same* confirmation token for both account numbers. The probability for such an event is $\frac{1}{10^n}$, with n being the number or relevant digits when we assume that account numbers are uniformly distributed.

Furthermore, an attacker could try to perform an illegitimate transaction to an account number that is very similar to the original account number entered by the user. However, our proposed scheme significantly raises the difficulty bar for the attacker because in order to launch a successful attack, he would have to own a similar account number *for each* account number the user enters.

Chapter 6

Implementation

We implemented the prototype of the ideas we proposed in the previous chapter to perform usability testing and to identify eventual weaknesses in our approach. An important design consideration was the integration of our prototype with existing web applications. Hence, we used common and popular web technologies such as the Common Gateway Interface (CGI). Our choice of the implementation language was guided by performance requirements as well as the availability of (graphical) open source libraries. Since interpreted languages such as Perl have a performance penalty when used with standard CGI (i.e., the overhead when launching and killing the interpreter for each request), we used compiled languages (i.e., C and C++). C was used for the most performance-critical parts, that is, CAPTCHA generation and grading. C++ was used for the more text-intensive parts (HTML code, SQL queries) due to superior string handling. We are aware that modern web servers like Apache [2] have built in script interpreters (`mod_perl, mod_php`) that perform en par with (compiled) CGI binaries, but we want to stress again that we aimed at developing a universally deployable solution and CGI is supported by virtually every web server.

6.1 The Framework

In order to evaluate our techniques on real users and to demonstrate that they are easy to implement and integrate with existing web applications, we built a simple proof-of-concept online banking web application that provides transaction functionality. That is, after a user logs in, she can perform (simulated) money transactions. Our transaction functionality is similar to real-life online banking web applications where the user is required to enter a target account number, a routing number and a sum that is to be transferred.

Screenshots of our web application are presented in Appendix A.
We follow the general structure of web applications. We have a database back end, where relevant user and session data is persistently stored. A CGI application generates the HTML code on-the-fly, fetching the required data from the database and incorporating eventual user input. On the client side, JavaScript is used to perform initial input validation and the client part of the CAPTCHA input protocol.
Each sub-page (login, choose method, enter data, enter confirmation) is generated dynamically. Normally, this would not be necessary, but we wished to keep a tight record on the users' actions for evaluation purposes. Therefore, each page is generated by the CGI application, and in the process, a time stamp and eventual submitted data is recorded in a "tracking table" in the database.
As discussed in the previous chapter, based on the technique the user wishes to use, either a CAPTCHA input element is presented for entering the input or confirmation tokens are requested after the input has been entered.

6.2 Confirmation Tokens

To realize the confirmation token schemes, we created simple scripts that would produce random token lists in printable form (i.e., LaTeX source files → PDFs) and at the same time enter the tokens into the database for automatic grading. For the lookup scheme, the tokens are similar to regular TANs (see Fig. 5.3). The application chooses the index and computes the page number just as the user would do, and retrieves the correct token from the database to compare it with the user-submitted token.
For the token computation scheme, a human readable version of the computation algorithm is printed, while a machine-readable (a Perl code snippet) is placed in the database (Fig. 6.1). Matching algorithms are identified by the same ID. When a submitted token is to be verified, Perl code representing the input values (e.g., the target account number) is generated on-the-fly and concatenated with the predefined algorithm snippet. In a second step, the resulting Perl program is executed using a built-in Perl interpreter (via `libperl`). Finally, the result of the operation is read into a string variable and compared with the submitted input. This mechanism allows for an easy extension or modification of the algorithm list. For our test runs, we created five algorithm classes and varied the significant positions and the constants, producing a list of ≈ 500 distinct algorithms.

```
Token ID 4:
  Human-readable:
    Create a number using the 5th and 7th
    digits of the target account and add 542 to it.
  Machine-readable:
    $v1 = substr $dst, 4, 1;
    $v2 = substr $dst, 6, 1;
    $code = ($v1 .  $v2) + 542;
```

Figure 6.1: The human and machine readable forms of a token computation algorithm.

6.3 The CAPTCHA Graphical Keypad (ImageIO)

Although there are many open source (and also non-free) CAPTCHA implementations available we did not find one we could use for our purposes. This is mainly due to the fact that no one thought about using CAPTCHAs the way we do and the capabilities of the CAPTCHA generators are rather constrained.[1] Specifically, they do not give sufficient control over the placement of characters in the image and retain only the complete string without any extra information for the grading comparison. Therefore, we decided to build our own CAPTCHA generator from scratch with all the features we required:

- Image generation
- Rendering of text
- Drawing of geometric shapes
- Image transformations
- Image output in web-compatible formats (e.g., PNG)
- Handling of HTTP requests & generating HTTP replies
- High performance

[1]On a side note, most of the available (even commercial) solutions are pretty poor by the standards we established in Chapter 4.

To meet the performance criterion, we wrote the generator in C and used two excellent open source C libraries to handle the HTTP part (`libcgic` [6]) and the image part (`libgd` [7]). The CAPTCHA generator is designed to be easily integrateable in any existing web application, that is, to replace a `<input type=‘‘text’’>` HTML element. Therefore, we made the interface as lean as possible.

6.3.1 ImageIO Generation

To produce an ImageIO CAPTCHA keypad, the generator creates two images with identical dimensions. In accordance with Figure 6.2, we will refer to them as left and right. The left image is the one displayed to the user, the right one is the lookup map stored on the server. The desired input elements, in our case the digits 0–9, are placed at random (but not overlapping) in the left image. The bounding boxes of the digits are rendered in the right image. The color of the bounding box encodes the value of the related input element.[2] Any additional text that is to be displayed (e.g., the reflected input) is rendered in the left image only. Then, miscellaneous transformations (i.e., rotate, scale, shear, warp, etc.) are applied to the left image and mirrored on the right. This guarantees that the input elements and their lookup counterparts are always congruent. Finally, clutter (salt-and-pepper noise and geometric shapes) is added to the left image.

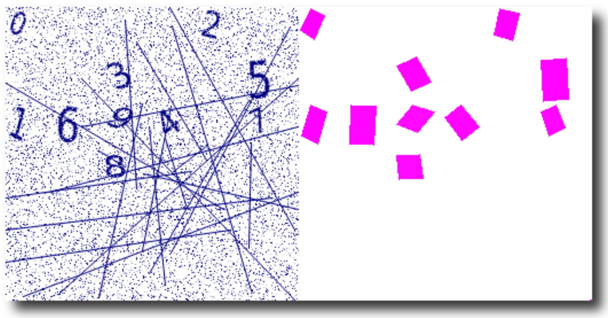


Figure 6.2: The CAPTCHA input element and the corresponding server-side lookup image side-by-side.

The lookup map is used to decipher the input the user wishes to submit. The map is sampled at the coordinates the user chose by clicking the input image and the color value determines what the user has clicked on. Using an

[2]We use 8-bit palette-based images as maps — that is sufficient to encode the entire ISO-8859-1 character set.

image instead of simply storing the coordinates of each bounding box' corners has two advantages: (a) Exact congruence of input element and bounding box, taking into account that after the transformations the box is in fact an irregular polygon. (b) Lookup times are $O(1)$. On the down side, there is the overhead from loading the image but with the small image sizes[3] in question we found it negligible.

We have to concede, that our prototype CAPTCHA is only mediocre when it comes to OCR resistance, but it was never an objective of this work to create the perfect CAPTCHA. Instead, our main goal was to test if the idea itself was acceptable to the users and thus, presenting them with a challenge of comparable hardness. The generator is designed to allow for easy replacement of the CAPTCHA rendering engine without affecting any other part, so stronger algorithms can be included should the need arise.

6.3.2 ImageIO Protocol

The protocol, that is, the parameters and possible values accepted by the ImageIO binary, is kept simple. The `action ID` tells ImageIO what action is requested. The two relevant[4] actions are `input` and `display`. Each `input` request generates a new ImageIO keypad and needs a second parameter, either `target` or `position`. With `target`, the name of the field we wish to provide input for is set. This allows us to have multiple ImageIO keypads on one page and is comparable to the `name` attribute of a `<input>` element which is used as parameter name when the corresponding form is submitted. When the `target` is set, each request with a `position` parameter (with the click coordinates as value) is treated as input to the active field. Other actions like clearing the input (after an eventual error) are realized in the same manner, that is, some positions in the image are reserved for "special actions," denoted by special characters.

The `display` action was added after initial usability tests. It generates a small image containing only the input received so far for the field denoted by the accompanying `target` parameter. This image can be displayed alongside other (regular) `<input>` elements in a form without breaking the layout and thus improving the legibility of the page, while the keypad resides in an unobtrusive location (or, as in our implementation, is freely positionable by the user).

[3]E.g., 256x256 pixel 8-bit paletted PNG $\approx$ 1KiB

[4]There are other actions implemented for debugging purposes.

Finally, to establish a connection between the different actions and also the rest of the web application, a random generated session ID is sent with each request.

6.3.3 ImageIO Client-side Implementation

In the client browser, JavaScript is used to register user clicks and generate the appropriate requests. The only (surprisingly) challenging part of that is obtaining the *correct* coordinates of the click event relative to the input image. There are many different user agents available and while the W3C has established standards for all the involved protocols and languages (HTML, CSS, JavaScript, DOM, etc.), the implementations by the browser manufacturers vary greatly. On different browsers, there are different ways to both calculate an objects position and an events position relative to the page origin (the difference of both being the events position relative to the object — this is what we need). Fortunately, many web designers have faced this dilemma before, and many helpful resources are available on the web.
This problem set aside, the remaining functionality is straight-forward. The requests to the ImageIO generator are created by changing the `src` attribute of the appropriate image (`input` or `display`). This method is available since the earliest JavaScript versions and is supported by virtually every browser.

6.3.4 Integrating ImageIO with a Web Application

A web application that needs to protect some input with ImageIO has to implement the following: Each page with ImageIO elements needs to include the JavaScript file implementing the client-side functionality. Placeholder images for `display` and `input` elements (with appropriate `id` and `class` attributes) are required, so the script functions can locate them. On the server side, for each session a configuration file (identified by the session ID) for the ImageIO generator is required, specifying the names of the protected fields and the dimensions of the images. After the user completes her input via ImageIO and submits the entire form, the securely transmitted value can be retrieved by the web application from another file created by the ImageIO generator (and also identified by the session ID). It would also be possible for ImageIO to leave the values in a database, but the amount of data in question is very small and the file-based solution is decisively faster.

6.4 Simulated Trojan Attack

We included a simulation of a Trojan horse attack in our testing framework. The rationale behind this will be explained in the next chapter. The task of the fake Trojan was to emulate the attack outlined in Section 2.2, that is, to modify the account number in a request sent to initiate a money transfer. The attack was implemented on the server side for three reasons: (a) We wanted to control when the attacks occurred and we wanted to assure each test session started with a few untampered transactions to avoid confusing the test subjects, (b) we had to record the occurrence of attacks in our tracking table (see Section 6.1) for later evaluation and (c) we faced the prospect of performing the tests on different machines and browsers, so the centralized solution was more economic.

Chapter 7

Evaluation

In Chapter 5, we discussed a number of possibilities an attacker possesses to subvert our protection schemes. The discussion showed that our system offers appealing security properties even in the presence of powerful adversaries. However, to be usable in practice, it is not sufficient that a scheme is secure, but it is also necessary that it is easy to understand and to use. In Section 2.4 we have established *usability* as one of the three fundamental properties of secure web applications. To assess the ease-of-use of our proposed protection mechanisms, we conducted a small-scale user study. This chapter summarizes the results of the study and presents our conclusions that outline differences in the acceptance by different user groups as well as necessary improvements to the user interface.

7.1 Usability Study Setup

The goal of the user study was to evaluate how suitable our protection mechanisms are to protect online bank transactions. Each participant in this study was asked to log into our prototype bank site (as described in Chapter 6) and carry out a number of money transfers. To this end, every user was provided with a list of transactions that s/he should perform; each entry contained the target account number, routing number, and the amount. Also, for each transaction, the user was told to select one of the three available protection schemes.

On the login page, a help section was displayed that briefly described all protection methods and the input data that the server required for a money transfer. This text was about two screen pages long and contained a description of the basic workflow. It had no figures or examples. In the beginning, users were given no additional information by the experimenter. That is, the task of the test subject was to work through the given list of transactions solely based on the description of the textual introduction. When a user failed to correctly perform a transaction and gave up on a certain protection method after some time, the experimenter took on the role of a customer support agent and was allowed to answer some specific questions or to address a particular problem that the subject encountered. For each protection technique, a number of transactions were performed. We silently observed the problems that subjects encountered before any hints were given, and also after some support was provided. At the end, we asked each person to rank the protection mechanisms with regards to their personal feeling of ease-of-use. Also, we were interested in suggestions that people might have to improve their user experience.

In addition to the ease-of-use, our study was also designed to assess the security of schemes that require some collaboration (and alertness) from users. To this end, we pretended that a Trojan horse was installed on the client machine. This Trojan horse attempted to tamper with some of the transactions. Of course, the malware could silently change the user input that is sent to the bank. However, our schemes are precisely designed to counter such attacks. Thus, we decided to design a Trojan horse that sends malicious data to the bank server *and* displays the modified account number openly on the page where the user has to confirm the entered information. The hope (from the point of view of the attacker) was that the user would accept the malicious account number and use the incorrect data for the calculation or lookup of the confirmation token. In case of the CAPTCHA-based solution, the Trojan horse had to rely on the fact that a user would not recognize that random points in the image were clicked (instead of the desired digits). We did not specifically advise users of the fact that the machine could be compromised. The introductory text, however, contained a warning that pointed out that the local machine could be infected with malware that might attempt to interfere with the bank transfers (similar to warnings found on real-life banking web sites).

In total, our study included 16 users. To ensure that the results of our study have some generalizable meaning (despite the small sample set), we attempted as much as possible to select participants from diverse backgrounds and different ages. For this study, we considered four different user attributes: age, education, technical sophistication, and previous experience with online banking. The *age* attribute was divided into three classes: less than 30, 30 to 50, and more than 50 years old. *Education* has only two values: technical and non-technical. These values are assigned based on the type of school that a person has attended. Note that technical education is not only limited to computer science, but includes all subjects that are related to mathematics (such as engineering or physics). We introduced this attribute to understand the effect of previous exposure and affinity to math on the ability to solve the computations required for confirmation tokens. The *technical sophistication* is an attribute with two values: yes and no; a user is considered technically sophisticated when she is able to install her own operating system and maintain it (e.g., by installing updates or device drivers). *Previous experience with online banking* also has two values, and should be self-explanatory. An overview of our subjects with their corresponding attribute values is shown in Table 7.1.

Total		16
Age	Less than 30	7
	30-50	3
	Older than 50	6
Education	Technical	10
	Non-technical	6
Tech. sophistication	Yes	7
	No	9
Exp. with online banking	Yes	13
	No	3

Table 7.1: Subjects of user study.

7.2 Usability Study Results

The 16 subjects performed a total of 189 transactions. This amounts to about four transactions per person for each of the three protection methods. The first transaction for each method was guaranteed to be free of any tampering from the simulated Trojan horse. This should ensure that a test subject does not attribute the effects of a malicious modification to a legitimate but misunderstood action required by the bank application. After the first transaction for a particular method, the simulated malware would tamper with all subsequent transactions with a probability of $\frac{1}{3}$. This relatively high probability was selected to ensure that most users would encounter attacks early on to be able to observe how they reacted to unforeseen events. Table 7.2 provides a summary of the transactions that were performed with each protection technique. This table lists the total number of transactions, the number of transactions that were maliciously modified (*bad transactions*), and the number of regular transactions (*good transactions*). For both the bad (Table 7.4) and the good (Table 7.3) transactions, we present tables that show a breakdown of how users reacted. That is, the tables list the number of times a transaction successfully completed, the number of times it was aborted, and the number of times it was unfinished. Note that a successfully completed bad transaction means that a Trojan horse would have managed to redirect the money transfer to an arbitrary account, despite any protection mechanism in place. An unfinished transaction is a transaction for which the user never provides a confirmation. This can have various reasons: the user pressed the back button and started a new money transfer, the browser crashed, etc.

	Total	Good	Bad
Graphical Input	50	37	13
Token Calculation	61	42	19
Token Lookup	78	62	16
All Techniques	189	141	48

Table 7.2: Transactions performed for user study.

	Total	Completed	Aborted	Unfinished
Graphical Input	37	30	2	5
Token Calculation	42	40	1	1
Token Lookup	62	53	3	6
All Techniques	141	123	6	12

Table 7.3: Regular transactions (the "Good")

	Total	Completed	Aborted	Unfinished
Graphical Input	13	5	4	4
Token Calculation	19	5	13	1
Token Lookup	16	2	10	4
All Techniques	48	12	27	9

Table 7.4: Simulated attacks (the "Bad")

It can be seen from Table 7.4 that 12 out of 48 attack attempts were successful. For us, this was a quite surprising result, as it indicated that in seven cases, people used the *incorrect* account number displayed by the Trojan horse to perform token lookup and token calculations. Also, in the five case where graphical input was used, the incorrect account number was clearly visible on screen, completely different from the digits that the user previously clicked. These 12 successful attacks underline in a disturbing fashion the limits of security solutions that require user cooperation. Our proposed detection techniques force an attacker to rely on social engineering to trick a user to accept a clearly incorrect account number. This is in contrast to current solutions, where the attack can be carried out completely transparently. Indeed, 36 malicious transactions were discovered and left unfinished. However, there was still a substantial fraction of users who could be deceived. To support these users, research into novel user-interface designs seem to be the most promising direction.

In general, we observed that, initially, most users had difficulties to understand how our protection mechanisms work. In particular, this was true for the algorithm calculation of the confirmation token and the CAPTCHA-based input. The mechanism to look up the code depending on the input was the easiest to understand. This is not surprising, as users familiar with online banking already have to perform a very similar task when using their real bank accounts (by looking up indexed TAN numbers). Nevertheless, after users got over the initial obstacles and had a chance to develop some familiarity with a new technique, they were typically able to carry out transactions swiftly and with few problems. When asked about the favorite protection mechanism, each user was supposed to assign three points to the preferred technique, two points to the second-best, and one point to the least-appreciated input mechanism. The method of looking up codes (*token lookup*) ranked first, with a total of 45 points. The second place was taken by the CAPTCHA-based approach (*graphical input*) with 27 points, closely followed by the algorithmic calculations of tokens (*token calculation*) with 24 points. For a complete overview of the assignments of points based on user attributes, refer to Table 7.5.

Table 7.5 shows that the token lookup method scores highest among all users. It is interesting that there is a negative correlation between the age and the appreciation for the graphical input. We assume that this is related to the problem of recognizing distorted digits in the CAPTCHAs. This hypothesis is supported by a number of comments of mainly older users that reported problems with distinguishing certain digits (such as ’1’ and ’7’). The strongest difference in preferences can be found among users with different technical sophistication. It seems that sophisticated users better understand the ad-

Algorithm	Age			Education	
	-30	30-50	50+	Tech.	Non-Tech.
Graphical Input	16	4	7	16	11
Token Calculation	8	5	11	15	9
Token Lookup	18	9	18	29	16

	Techn. Sophistication		Online Banking	
	Yes	No	Yes	No
Graphical Input	16	11	22	5
Token Calculation	8	16	20	4
Token Lookup	18	27	36	9

Table 7.5: Preference points for different protection techniques.

vantages of graphical input and appreciate the fact that “they don’t have to get up and look for the code book.” Users that have less experience with computers, on the other hand, seem to prefer the token calculation, which is closer to the traditional TAN system used by online banking today. Finally, note that a technical education does not favor the use of the token calculation.

As stated previously, we observed that users were able to use our protection mechanisms with few problems once the process was understood. Of course, part of the complexity lies in the way our techniques are designed. However, a non-negligible part of the initial difficulties can also be attributed to weaknesses in our user interface design. For example, we only provided a textual explanation of the protection mechanisms. Obviously, many users simply refused to read the online manual and failed later. For them, a simple example that (interactively) guides through the process would have been helpful. Another issue was expressed by a user with a PhD in mathematics, who criticized the descriptions of our algorithms as too imprecise. Indeed, this was the expression of a general feeling many users shared. For example, for one algorithm, we stated that the result of a calculation should be “concatenated” with some value. Some people misinterpreted the concatenation operation as summation and added the value to the previous result. Also, it was often unclear what to do if one part of the calculation yielded zero. For the graphical input, some users attempted to type the CAPTCHA input on the keyboard, especially the `delete` and `next` symbols. Another problem with the graphical input was that it was not trivial to correct an input error that occurred some digits ago. Our current implementation requires that all input up to the erroneous position is deleted before the problem can be resolved.

This frustrated some users.
Given the results of our evaluation and our experiences while conducting the user study, it was apparent that once our proposed solution was understood, most people were able to perform the required steps with little difficulty. Thus, considering the additional protection that our techniques provide, we believe that they are suitable for deployment in security-critical environments such as online banking. Also, given the different preferences of different user groups, one can envision a scenario in which a bank offers multiple protection techniques in parallel and let the user select her favorite one.

Chapter 8

Conclusion and Future Work

Web applications have become the most dominant way to provide access to online services. A growing class of problems are client-side attacks in which malicious software is automatically installed on the user's machine. This software can then easily access, control, and manipulate all sensitive information in the user's environment. Hence, an important web security research problem is how to enable a user on an untrusted platform to securely transmit information to with a web application.

Traditional TAN-based schemes are vulnerable to Trojan and phishing attacks. Previous solutions to this problem are mostly hardware-based and require peripheral devices such as smart card readers and mobile phones. Such solutions are often expensive and not always available to users. Software-based solutions, such as graphical keyboards were promising at first, but some were poorly implemented, providing a false sense of security, and the good ones were beaten with screenscrapers and OCR techniques.

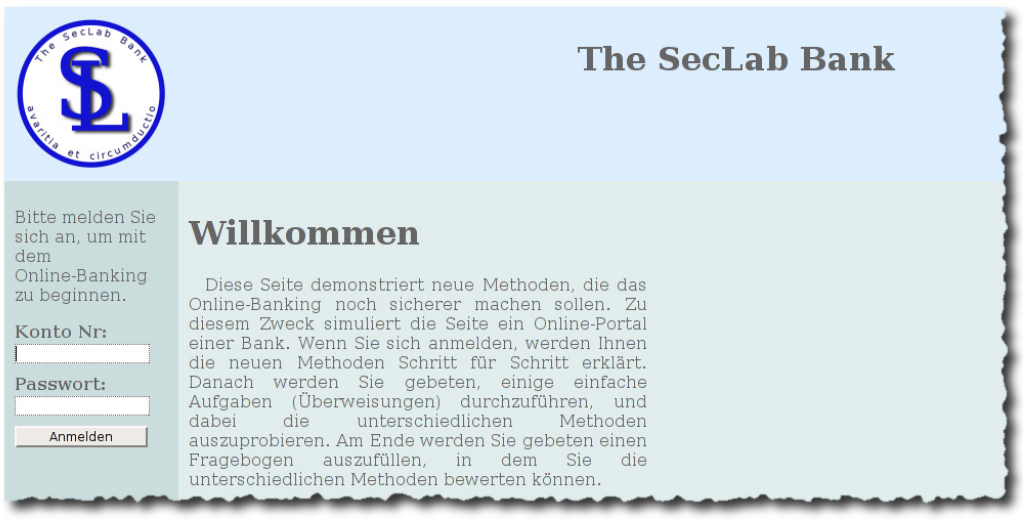
In this thesis, we present two novel server-side techniques that can be used to enable secure user input. The first technique extends graphical input with CAPTCHAs to protect the integrity of user input against automated attacks. We have discussed the current uses of CAPTCHAs and provided arguments that it is possible to create OCR-resistant CAPTCHAs, making this idea feasible. The second technique uses confirmation tokens that are bound to sensitive data to ensure data integrity. Confirmation tokens can either be looked up directly in a code book or they need to be calculated using simple algorithms.

The usability study that we conducted demonstrates that, after an initial learning step, our techniques are understood and can also be applied by a non-technical audience. The experiences gathered from that study have allowed us to improve our solutions in order to make them more usable. We are planning to conduct further usability studies to evaluate these improvements.

Our dependency on the web will surely increase in the future. At the same time, client-side attacks against web applications will most likely be continuing problems as the attacks are easy to perform and profitable. We hope that the techniques we present in this thesis will be useful in mitigating such attacks.

Appendix A

Screenshots of our Simulated Online-banking Site



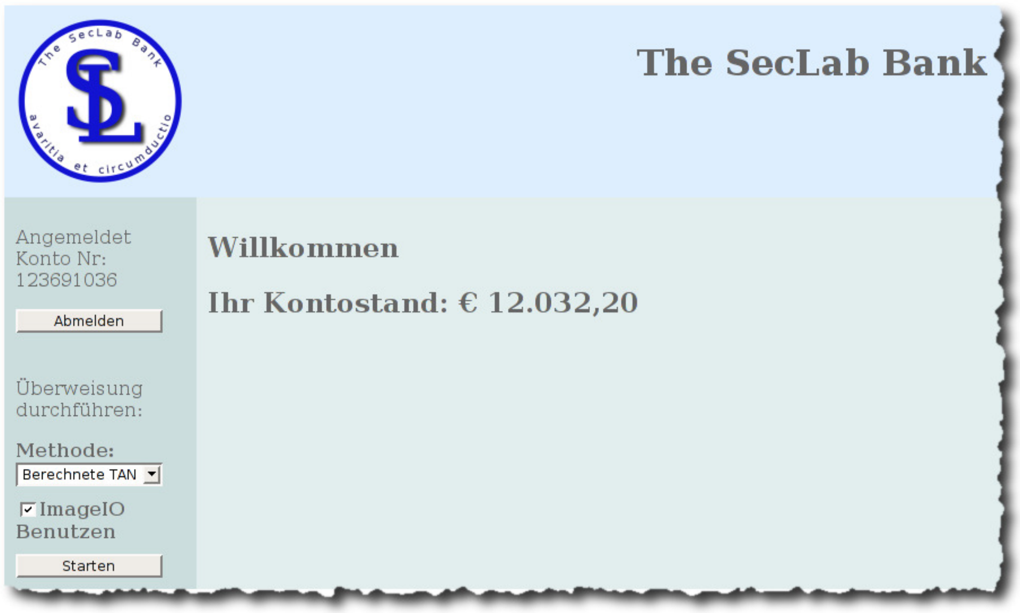
The SecLab Bank

Bitte melden Sie sich an, um mit dem Online-Banking zu beginnen.

Willkommen

Diese Seite demonstriert neue Methoden, die das Online-Banking noch sicherer machen sollen. Zu diesem Zweck simuliert die Seite ein Online-Portal einer Bank. Wenn Sie sich anmelden, werden Ihnen die neuen Methoden Schritt für Schritt erklärt. Danach werden Sie gebeten, einige einfache Aufgaben (Überweisungen) durchzuführen, und dabei die unterschiedlichen Methoden auszuprobieren. Am Ende werden Sie gebeten einen Fragebogen auszufüllen, in dem Sie die unterschiedlichen Methoden bewerten können.

Figure A.1: The login page


The SecLab Bank

Angemeldet
Konto Nr:
123691036

Überweisung
durchführen:

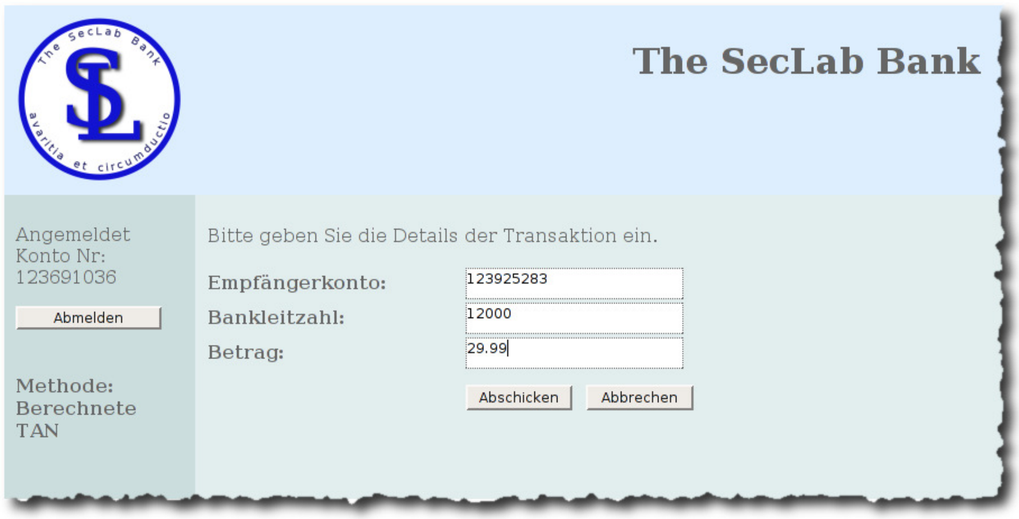
Methode: Berechnete TAN

[✓] ImageIO Benutzen

Willkommen

Ihr Kontostand: € 12.032,20

Figure A.2: Method selection


The SecLab Bank

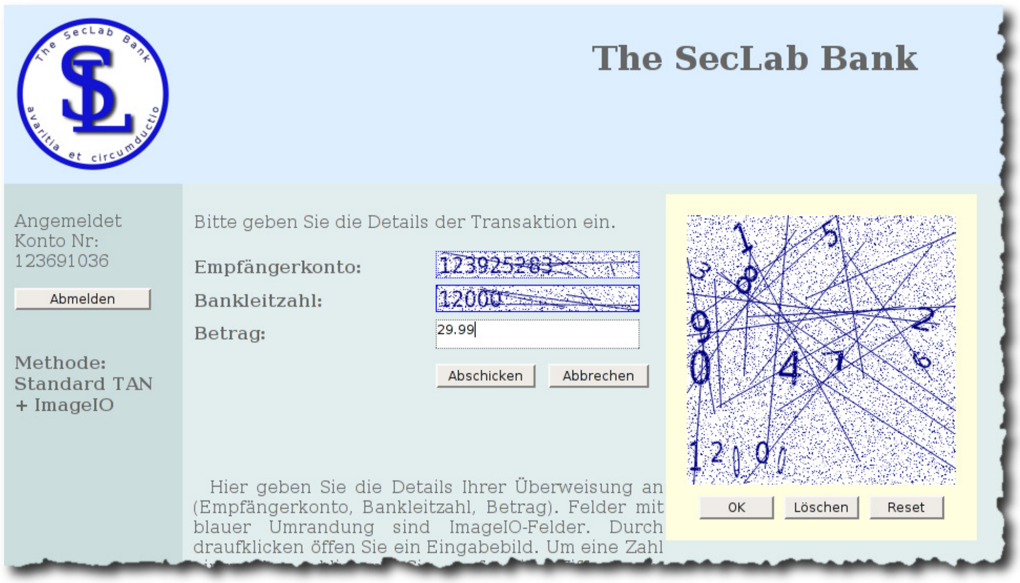
Angemeldet
Konto Nr:
123691036

Methode:
Berechnete
TAN

Bitte geben Sie die Details der Transaktion ein.

Empfängerkonto:	123925283
Bankleitzahl:	12000
Betrag:	29.99

Figure A.3: Form for transaction details


The SecLab Bank

Angemeldet
Konto Nr:
123691036

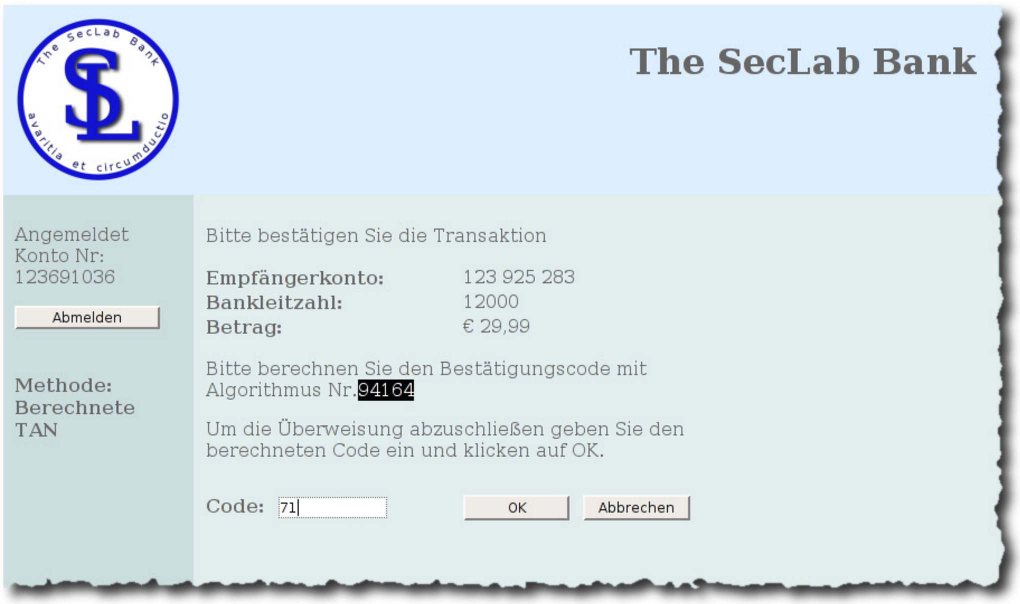
Methode:
Standard TAN
+ ImageIO

Bitte geben Sie die Details der Transaktion ein.

Empfängerkonto:	123925283
Bankleitzahl:	12000
Betrag:	29.99

Hier geben Sie die Details Ihrer Überweisung an (Empfängerkonto, Bankleitzahl, Betrag). Felder mit blauer Umrandung sind ImageIO-Felder. Durch draufklicken öffen Sie ein Eingabebild. Um eine Zahl

Figure A.4: Form for transaction details with CAPTCHA input element


The SecLab Bank

Angemeldet
Konto Nr:
123691036

Methode:
Berechnete
TAN

Bitte bestätigen Sie die Transaktion

Empfängerkonto:	123 925 283
Bankleitzahl:	12000
Betrag:	€ 29,99

Bitte berechnen Sie den Bestätigungscode mit Algorithmus Nr. 94164

Um die Überweisung abzuschließen geben Sie den berechneten Code ein und klicken auf OK.

Code: 71

Figure A.5: Form for confirmation token — in this case it's the "token calculation" variant

Bibliography

[1] Anti-phishing Working Group. http://www.antiphishing.org.

[2] Apache Software Foundation. HTTP Server Project. http://httpd.apache.org.

[3] Presseinformationstelle Baden-Württembergische Bank. TAN-Generator der BW-Bank zertifiziert - Erfolg für sicheres Onlinebanking. http://www.bw-bank.de/bwbankde/1000005999-s1463-de.html, Dec 2006. (In German).

[4] Henry S. Baird, Michael A. Moll, and Sui-Yu Wang. ScatterType: A Legible but Hard-to-Segment CAPTCHA. In *ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 935–939, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Dirk Balfanz and Ed Felten. Hand-Held Computers Can Be Better Smart Cards. In *Proceedings of the 8th Usenix Security Symposium*, 1999.

[6] Thomas Boutell. cgic: an ANSI C library for CGI Programming. http://www.boutell.com/cgic/.

[7] Thomas Boutell and Pierre Joye. GD Graphics Library. http://www.libgd.org/.

[8] Carnegie Mellon University. The CAPTCHA Project. http://www.captcha.net, 2000.

[9] CERT. CERT Advisory CA-2001-26 Nimda Worm, 2001.

[10] Kumar Chellapilla, Kevin Larson, Patrice Y. Simard, and Mary Czerwinski. Building Segmentation Based Human-Friendly Human Interaction Proofs (HIPs). In Henry S. Baird and Daniel P. Lopresti, editors,

HIP, volume 3517 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2005.

[11] Kumar Chellapilla, Kevin Larson, Patrice Y. Simard, and Mary Czerwinski. Computers beat Humans at Single Character Recognition in Reading based Human Interaction Proofs (HIPs). In *CEAS*, 2005.

[12] Kumar Chellapilla, Kevin Larson, Patrice Y. Simard, and Mary Czerwinski. Designing human friendly human interaction proofs (HIPs). In Gerrit C. van der Veer and Carolyn Gale, editors, *CHI*, pages 711–720. ACM, 2005.

[13] Kumar Chellapilla and Patrice Y. Simard. Using Machine Learning to Break Visual Human Interaction Proofs (HIPs). In *NIPS*, 2004.

[14] William Cheswick. Johnny Can Obfuscate: Beyond Mother's Maiden Name. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HotSec)*, 2006.

[15] Monica Chew and J. D. Tygar. Image Recognition CAPTCHAs. In Kan Zhang and Yuliang Zheng, editors, *ISC*, volume 3225 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 2004.

[16] Neil Chou, Robert Ledesma, Yuka Teraguchi, and John C. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of the Network and Distributed Systems Security (NDSS)*, 2004.

[17] Ritendra Datta, Jia Li, and James Z. Wang. IMAGINATION: a robust image-based CAPTCHA generation system. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 331–334, New York, NY, USA, 2005. ACM Press.

[18] eWeek.com. Trojan Redirector Ups the Ante in Online Banking Attacks. `http://www.eweek.com/article2/0,1895,1940623,00.asp`, Mar 2006.

[19] Rony Ferzli, Rida Bazzi, and Lina J. Karam. A CAPTCHA Based on the Human Visual System Masking Characteristics. In *IEEE International Conference on Multimedia & Expo (ICME)*, Jul 2006.

[20] FinExtra.com. Phishers move to counteract bank security programmes. `http://www.finextra.com/fullstory.asp?id=14149`, 2005.

[21] John Graham-Cumming. Anti-Spam Tool League Table. `http://www.jgc.org/astlt.html`.

[22] Samuel Hocevar. PWNtcha - Captcha Decoder. `http://sam.zoy.org/pwntcha`.

[23] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley, 2005.

[24] IETF Working Group. Transport Layer Security (TLS). `http://www.ietf.org/html.charters/tls-charter.html`, 2006.

[25] International Organization for Standardization (ISO). ISO 7816 Smart Card Standard. `http://www.iso.org/`.

[26] Collin Jackson, Dan Boneh, and John C. Mitchell. Stronger Password Authentication Using Virtual Machines. `http://crypto.stanford.edu/SpyBlock/spyblock.pdf`.

[27] Auson Josang, Dean Povey, and Anthony Ho. What You See is Not Always What You Sign. In *Annual Technical Conference of the Australian UNIX and Open Systems User Group*, 2002.

[28] M. D. Lillibridge, M. Abadi, K. Bharat, , and A. Broder. Method for selectively restricting access to computer systems. US Patent 6,195,698, Applied April 1998 and Approved February 2001.

[29] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Bump in the Ether: A Framework for Securing Sensitive User Input. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[30] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, chapter 11, pages 425–488. CRC Press, Oct 1996.

[31] Microsoft Anti-Malware Engineering Team. Windows Malicious Software Removal Tool: Progress Made, Trends Observed. `http://go.microsoft.com/fwlink/?linkid=67998`, Jun 2006.

[32] Microsoft Corporation. Internet Explorer 7 features. `http://www.microsoft.com/windows/ie/ie7/about/features/default.mspx`.

[33] Greg Mori and Jitendra Malik. Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Conference (CVPR).* IEEE Computer Society Press, 2003.

[34] Shunji Mori, Ching Y. Suen, and Kazuhiko Yamamoto. Historical review of OCR research and development. *Document image analysis*, pages 244–273, 1995.

[35] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A Crawler-based Study of Spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[36] MSNBC.com. A new, more sneaky phishing attack. `http://www.msnbc.msn.com/id/6416723/`, Nov 2004.

[37] Moni Naor. Verification of a human in the loop or Identification via the Turing Test. Unpublished Manuscript, `http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human.ps`, 1997.

[38] Siani Pearson. *Trusted Computing Platforms*. Prentice Hall, 2002.

[39] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger Password Authentication Using Browser Extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[40] Amalia Rusu and Venu Govindaraju. Handwritten CAPTCHA: Using the Difference in the Abilities of Humans and Machines in Reading Handwritten Words. In *IWFHR '04: Proceedings of the Ninth International Workshop on Frontiers in Handwriting Recognition (IWFHR'04)*, pages 226–231, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Joanna Rutkowska. Introducing Blue Pill. `http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html`, 2006.

[42] Secure Information Technology Center Austria (A-SIT). The Austrian Citizen Card. `http://www.buergerkarte.at/index_en.html`, 2005.

[43] The SecuriTeam. Defeating Image-Based Virtual Keyboards and Phishing Banks. `http://blogs.securiteam.com/index.php/archives/678`, 2006.

[44] John F. Shoch and Jon A. Hupp. The "worm" programs — early experience with a distributed computation. *Commun. ACM*, 25(3):172–180, 1982.

[45] silicon.com. Hackers nab € 800,000 in online banking attack. `http://software.silicon.com/security/0,39024655,39165310,00.htm`, Jan 2007.

[46] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical Report Purdue Technical Report CSD-TR-823, Departament of Computer Science, Purdue University, West Lafayette, IN 47907-2004, 1988.

[47] Adrian Spalka, Armin Cremers, and Hanno Langweg. Protecting the Creation of Digital Signatures with Trusted Computing Platform Technology Against Attacks by Trojan Horse. In *IFIP Security Conference*, 2001.

[48] Adrian Spalka, Armin Cremers, and Hanno Langweg. Trojan Horse Attacks on Software for Electronic Signatures. *Informatica*, 26, 2002.

[49] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[50] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using Hard AI Problems for Security. In *EUROCRYPT*, pages 294–311, 2003.

[51] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[52] W3C Working Group. Inaccessibility of CAPTCHA, Alternatives to Visual Turing Tests on the Web. `http://www.w3.org/TR/turingtest/`.