İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

DEVELOPING SECURE SOFTWARE WITH C AND C++: A DIFFERENT APPROACH

MS THESIS BY
Mehmet Barış SAYDAĞ
504021509

Date of submission: 12 August 2005
Date of defence examination: 1 September 2005

Supervisor (Chairman): Prof. Dr. Bülent ÖRENCİK
Members of the Examining Committee: Prof. Dr. Hasan DAĞ
Asst.Prof.Dr. Zehra Çataltepe

SEPTEMBER 2005

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

C VE C++ İLE GÜVENLİ YAZILIM GELİŞTİRME: FARKLI BİR YAKLAŞIM

YÜKSEK LİSANS TEZİ

Mehmet Barış SAYDAĞ
(504021509)

Tezin Enstitüye Verildiği Tarih: 17 Ağustos 2005
Sınav Tarihi: 1 Eylül 2005

Tez Danışmanı: Prof. Dr. Bülent ÖRENCİK
Diğer Jüri Üyeleri: Prof. Dr. Hasan DAĞ
Yrd. Doç.Dr. Zehra ÇATALTEPE

EYLÜL 2005

Preface

I always considered security as a very important aspect of software engineering process. This thesis has been an opportunity to share my research and knowledge with other academicians, developers and decision makers, who involves during different phases of software lifecycle.

With this opportunity, I would like to thank my supervisor Professor Doctor Bülent Örencik for supporting me with his vast experience and motivation while I was preparing this work.

08/12/2005 Mehmet Barış Saydağ

Table of Contents

Index of Figures

Index of Tables

Abbreviations

DCR: Design Change Request
GDR: General Distribution Release
DLL: Dynamic Link Library
ISP: Internet Service Provider
MITM: Man In The Middle
SQL: Structured Query Language
CRT: C Runtime
RPC: Remote Procedure Call
OSI: Open Systems Interface
MAC: Message Authentication Code / Medium Access Code
IV: Initialization Vector
STL: Standard Template Library
API: Application Programming Interface
GUI: Graphical User Interface
ATL: Active Template Library
MFC: Microsoft Foundation Classes
COM: Component Object Model
ODBC: Open Database Connectivity
UUID: Universally Unique Identifier
TCB: Trusted Computing Base
ISAPI: Internet Services Application Programming Interface. An API for writing custom extensions for IIS, web server of Windows platform.

Clarifications of Definitions

NULL: It is defined as 0 (zero) and used to describe value of zero.
NUL: It is used to describe the character, which on the 0^{th} position in ASCII table and used as termination sentinel at the end of C style strings (C++ style string are considered to be objects of **`string`** class.).

C ve C++ İle Güvenli Yazılım Geliştirme: Farklı Bir Yaklaşım

Ağa bağlı bilgisayarlar yaygınlaştıkça, günlük işlerin yürütülmesinden devlet sistemlerinin otomasyonuna kadar her seviyede rol almaya başlamışlar ve bu sistemlerin güvenliği de kritik bir hal almıştır. Bilgi işlem sistemlerinin güvene layık olabilmesi için bütün bileşenlerinin güvenli olması gerekir; yazılım da bu bileşenlerden birisi, belki de en önemlisidir. Yazılımların, yaşam süreçlerinin her aşamasında güvenli bir yapıyla sonuçlanacak şekilde tasarlanmaları ve geliştirilmeleri gerekmektedir.

Bu tez, bir yazılımın yaşam sürecini baştan sona ele almış ve getirdiği yeni fikirleri bu sürecin aşamalarına yerleştirmiştir. Konu ile ilgili yeterli arka plan bilgisi verdikten sonra yeni düşünceler tanıtılmış, örnekler verilmiş ve olabilecek başka seçenekler tartışılmıştır. Çoğu konuyu anlatırken, tamamlayıcı özelliği olduğu düşünülen bilgiler de ya tazin içinde verilmiş, ya da referans edilmiştir. Bu sayede geliştirme veya bakım gibi değişik aşamalardaki projelere referans kaynağı olarak hizmet verebilmektedir. Bu tezde ele alınan yaşam süreci, yazılım mühendisliğinde sıklıkla başvuru olarak kullanılan, süreci isteklerin tanımı, tasarım, geliştirme, kontrol etme ve bakım olarak bölümleyen "Şelale Yaşam Süreci"dir.

Yeni nesil programlama dilleri çıktıkça, C/C++ ve Birleştirici gibi düşük seviye dillerin yeni öğrencilerce benimsenmesi azalmaktadır. Buna ve başka sebeplere de bağlı olarak bu dillerde tecrübeli eleman eksikliği baş gösterdikçe, zaten güvenliğin sağlanmasının göreceli olarak daha zor olduğunun görüldüğü bu ortamlarda ciddi güvenlik açıkları oluşmaktadır. Dünya üzerindeki kod tabanının çoğunluğunun halen bu dillerden oluşması, durumu daha da kritik yapmaktadır. Bu makalede bahsedilen konuların çoğunluğu dilden bağımsız olsa da, ilgili bölümlerde, az önce bahsedilen sorunu göz önüne alarak C/C++ ve Birleştirici dilleri üstünde durulmuştur.

Sonuç olarak, yazılım güvenliğinin etkin olarak sağlanabilmesi için, güvenliğin bütün yaşam süreci evrelerinde ele alınması gerekliliği gösterilmiştir. Ayrıca, yaşam sürecinin aşamalarından bir çoğuna, daha önce bu kapsamda uygulanmamış olan yeni yöntemler önerilmiştir.

Developing Secure Software with C and C++: A Different Approach
SUMMARY

As networked computing penetrates daily life more and more, it becomes more common in every level from daily life to automation of government systems. In order computing systems to be secure, each and every of their components must be secure, too. Software is most important component among those. Each phase of software lifecycle must be implemented in a secure fashion.

This thesis is inspecting lifecycle of software from beginning to the end and aligns the new ideas that it is bringing to the lifecycle. After giving necessary background information about the subject, new ideas have been presented, examples have been given and possible other options have been discussed. During explaining most of the subjects, the topics that is considered to be complimentary is either added or referred to. Thanks to that, this thesis can be a reference source to projects in different phases like implementation and maintenance. Waterfall lifecycle model, which is used frequently in software development projects and divides software projects into phases as analysis of requirements, design, implementation, verification and maintenance, is used as a template in this thesis.

As new generations of programming languages emerge, adoption of low-level languages such as C/C++ and assembly by new students is decreasing. As lack of experienced staff shows up itself due to this and other causes, severe vulnerabilities are happening in such environments, where developing of secure software is already proven to be hard. The fact that majority of current code base in the world is in those languages makes the situation even more critical. Although most of the subjects in this thesis are programming language independent, C/C++ and assembler language problems are especially covered because of the reasons just mentioned.

As a result, it has been shown that security countermeasures must be taken in all phases of software lifecycle in order to ensure high level of security throughout the application. Furthermore, new ideas of security countermeasures have been brought to many of the phases of software lifecycle.

1. Introduction

1.1. Motivation

1.1.1. Connectivity Is Important

Motivation of this thesis comes from the fact that connectivity gets more important everyday and as it becomes an infrastructure for quality of daily life; its trustworthiness becomes a more important aspect.

With globalizing economy, business-to-business relationships extend beyond horizons and require high level of connectivity. With different time zones, there is always daylight in one corner of the world, keeping servers and applications busy 7/24. Even shortest amount of downtime causes big financial losses and damage to reputation. Trade secrets and sensitive information of customers reside in servers those host millions of connections from different (generally unauthenticated and thus anonymous) sources. Laws adapt to connectivity era, as well; there are severe penalties for irresponsibility of companies resulting in privacy reveal and identity thefts.

With improved battery life and accepted mobile communication standards, manufacturers provide mobile devices that offer seamless and continuous connectivity to the Internet. Lower priced devices come everyday with richer feature sets attracting more and more people to be connected. As these devices become a part of life, users depend on them more and more; high levels of robustness and reliability are requested even from basic, entry-level devices. Carrying those devices always with themselves causes storing increasing amount of private data in those devices; privacy protection becomes essential.

On the sharper edge of technology, people are known only with their digital identities; namely their email addresses, domain names, certificates, nicknames. Theft of this information turns into identity theft, allowing attacker to impersonate innocent and honest people for their illegal activities.

1.1.2. There Are New Challenges

In this connectivity era, hardware and software face new challenges. New challenges require new practices and disciplines. On the hardware side, improvements can be done more easily. Almost all metrics those define a high quality system can be achieved by just spending more money. Dependability can be sustained by buying redundancy, which actually ensures high amount of robustness. Security can be enforced easily, too; access to hardware is usually limited with regulated access to system rooms. Scalability, performance, responsiveness; virtually all of them can be acquired by buying more from off-the-shelf components; there is no hard needs for trade-offs.

Software has harder time to take these new challenges. There is no such thing like software redundancy; a program is whether running or not. Software is much more complex, programs do not have common interfaces, and each one solves completely different set of problems. Software is a thought, an idea; it is harder to understand, visualize and comprehend. Translation of thought into reality is very hard to measure and verify. Because it is impossible to see it, it is also impossible to see byproducts of it. All these traits make it harder to implement securely. Unfortunately, software security and reliability, thus trustworthiness, cannot be bought with just spending more money.

Up time is very important, attackers hit with denial of service attacks. Privacy is very important, attackers hit with network sniffing, man in the middle attacks, backdoors (using private APIs) and traffic analysis. There is a new attack with different attack vectors everyday, and often with brand new methods and tools. This thesis aims to provide developers with new set of information to enable even more secure software design and implementation, which helps their brainchild to withstand those (may be yet unknown) attacks

1.1.3. Software Must Be Secure

All software applications must be secure and trustworthy, because their usage areas, their lifetime and motivation of attacker cannot be known in advance. A designer thinking as “nobody would bother attacking this software” is most likely in huge mistake, because there might be people who is using that software for security

critical tasks in the threat of attackers any time in the future, well beyond expectations of developers.

1.2. Definition of the Problem

Defective code is a piece of code that is not doing its function properly. For instance, it is supposed to add two numbers and return the sum; however, it is returning the sum incorrectly. On the other hand, security issues are byproducts of otherwise perfectly healthy system. Code does more than what it should do. It is possible to say that, if it would be guaranteed that nobody would ever exploit a given security vulnerability, it is completely harmless and does not effect correct operation of the system. Then it would be left unfixed. Since they are byproducts, developers and tester should use their imaginations to discover what that byproduct would be. This is what makes secure software development so difficult. On the other hand, attackers must use their imagination, too. This is what makes never-before-seen creative exploits possible.

Software development technology is a rather new discipline if compared with other disciplines like mechanical engineering or civil engineering. Expectations from this discipline are advanced very fast. This discipline is now under demand of providing very high quality and security to never foreseen amount of people. Immature technologies and unprepared systems pose threats to consumers, which is in this case millions of people. Threats scale from minor inconveniences to serious reliability and security issues that are in prime time news almost weekly.

Generally speaking, software projects are consumed by far more people than any materialistic project (bridges, skyscrapers, space shuttle) in the given time span, because it is globally accessible. A single vulnerability leveraged by hackers can incapacitate certain tasks in Internet environment globally. Although software security threats are not (yet) safety threats like in materialistic, their wide applicability justifies efforts in investigation of security countermeasures.

Especially after year 2000, major software houses have taken important actions to prevent attacks, which damages their customers (as persons and as economic entities), public in broader perspective and, of course, themselves (as bad reputation). Academics started to invest more resources in software security as well, to serve

community and technology. Unfortunately, limited time of five years was not sufficient and there is still high volume of work that has to be done.

There has been different approaches to this subject; cookbook style plug-and-play solutions, in-depth analysis of just one concept like correct usage of C/C++ “const” qualifier are all examples. We think that although all of these researches are valuable previous work, they lack an important factor: Harmony and fit into the software development lifecycle. Since software is a product, and final outcome depends on the processes while bringing it to existence, engineering lifecycle is a very important aspect.

1.3. Purpose of this Thesis

The purpose of this thesis is stating that focus of security is important in every phase of software engineering lifecycle and security vulnerabilities are evitable if correct countermeasures are taken.

This thesis makes people of different roles aware of most severe and common errors that can cause security vulnerabilities. Another goal is providing them with a good reference of what to pay attention when developing trustworthy applications.

What distinguishes this thesis from other works previously done about this subject is approach to the subject and the new topics that is novel and not covered elsewhere. These differentiating factors are explained in the following sections.

1.3.1. Approach to the Subject

Designing and implementing high quality software, like every successful engineering project, is accomplished with well-defined and monitored methods. Although phases usually overlap and iterated, lifecycle of software can be defined basically as following phases:

- Analysis and definition of requirements,
- Planning,
- Implementation,
- Verification,
- Deployment and

- Maintenance

Security can (and should) be improved in each of these phases, though it is much more effective if taken account during earlier phases. This document devotes a chapter to each of these parts to stress what to do and not to do in that specific phase of lifecycle.

Although this engineering lifecycle is taught in almost every software engineering book, (most popular and widely known works include [12] and [13]. [43] is an online article which gives simple overview. [44] is an interesting debate of usage of a methodology at all.) and is referred in vast number of articles, unfortunately, none of these works give sufficient amount of information about security aspects of each steps. *This trait makes this thesis unique among other works.*

There are numerous works about security improvements of software projects; however, these works have different organizations than this thesis. I think that organizing principles in the way of this thesis is more natural, because people think of software lifecycle conceptually in that way. In even moderate sized software projects, different people take different responsibilities. Formal organization of work and responsibilities allows an easy ramp-up for a person who is joined to the team recently. It is widely known that new hires pose a high level of security threat because of their familiarities and inexperience with the project. Unique organization of this work aims reducing, if not completely preventing at all, this actually evitable threat.

There can be criticism about rather limited usage and simplicity of waterfall method. It is true that waterfall method is normally too simple to use in larger scale projects' management and engineering. Other more sophisticated methods (CMM, iterative, spiral, to name just a few) are generally preferred over waterfall method. However, we strongly believe that waterfall method is most natural and easiest to conceptualize method of software engineering. Moreover, other methods can easily be abstracted as functions of waterfall methodology, which gives that process a universal identity that makes it even more important. One last argument can be that developing secure software should be thought during education of students; and students (even new graduate hires) mostly use waterfall method.

1.3.2. New Topics

This thesis brings novel ideas on some topics those have not been published previously. In this thesis, there is some information that was available before this work; however, this information is presented to establish completeness or to give enough background information to base topics on.

◊ Design Patterns

Design patterns are widely used in moderate to large-scale software projects. "Gang of Four" has presented a very high quality of work [14] to categorize, define and discuss most popular and useful design patterns. This work was published back in 1995, before critical threads of attacks and companies' initiatives for secure software. Therefore, it lacks information about security during usage of those patterns. This thesis has a goal to cover this absent information by examining those patterns in that point of view. Since design patterns are generally used by moderate to large scale projects and security vulnerabilities mostly occur in that size of projects, this research presents a good deal of usability in practice, as well.

◊ Catalog of One Line Code Defects

Unfortunately, even a single defective code line can pose severe security vulnerability that can render whole software into an unreliable, untrustworthy and therefore unusable application. Moreover, if this vulnerability is taken advantage of with a successful attack, it can result in financial loses and privacy damage. Sadly enough, one defective line can have very bad consequences.

Obviously, software consists of code, which consists of lines. Therefore, preventing vulnerabilities at that level is a good start. Of course, there are bugs that is much complicated than simple one liners, but that is other topic. All simple bugs must be removed, since it is doable with several methods.

Although several previous works examines samples of those defects, none of them are complete. For example, Writing Secure Code [15] has focused information on vulnerabilities of only one defective code line; however, it is not a catalog, it misses some defects those are actually very common, too. One more drawback of this book is that it covers mostly Microsoft Corporation technologies, which can frustrate readers that use other technologies. Code Complete [16], on the other hand, has less

focused, very distributed coverage on that topic. Giving a good theoretical background but lacking of practical samples and focus, it has its own place.

Moreover, this thesis defines new examples of possible errors along with respective preventions. Novel subjects and pragmatic results make this thesis different in that perspective, too.

◊ Usage of Cryptographic Algorithms in Secure Software

Applied Cryptography [6] makes an outstanding job describing cryptographic algorithms and protocols. Unfortunately, that book is not written for software development in mind, therefore it lacks some important information, especially about application of cryptography and software lifecycle in commercial products. Secure Programming Cookbook for C and C++ [18] has practical applications, however it is based on a novelty API and it lacks theoretical information. Some information can be used as a "cookbook way", however, this black box plug-and-play approach is unsuitable for most of the serious big projects. This thesis, on the other hand, covers an important area of cryptography with enough level of theory and its application to secure software. That area is "cryptographic modes", which is very important during application of cryptography.

There are two reasons why this thesis is focused on this subject among others of such a broad domain as cryptology. First, a brilliant algorithm can be rendered useless and/or insecure with an unsuitable mode. Second, algorithms are developed after long research of academicians and there are readily available implementations accessible through operating system APIs. However, choosing an encryption mode is generally left to designers and developers. They are normally next big decision after choosing an encryption algorithm.

Besides modes, other aspects of design decisions like encryption method (stream versus block), compression and general principles are mentioned as well. Although that information can also be found on several other works, they are included here for the sake of completeness.

◊ COM Encapsulation as a Security Countermeasure

COM is a technology that is invented by Microsoft Corporation in late eighties and is one of the core functionalities of Windows OS. It helps encapsulation of

functionalities in separate binaries, in an advanced way than DLL's do. It is a well-studied subject from implementation and usage point of view. However, COM technology can be an instrument that enables software designers to vision more secure software. This thesis brings another implementation detail for least privileged user principle, which is not covered elsewhere. There are numerous books about COM, and generally, these books cover COM security, as well. Nevertheless, least privileged user account principle is different from COM security. COM security is user authentication to access to the services that COM module serves. Least privileged user account, on the other hand, is a design decision to encapsulate tasks into user contexts to minimize attack surface and related vulnerability.

◊ Libraries and their usage are investigated.

Developers use libraries to increase code reuse and cut from development time. Taking advantage of existing functionality is good idea unless that functionality does not bring its security threats with it. There is a saying that goes, as "Being able to ask is half of knowing." If developer is not aware of the potential vulnerabilities in the libraries that is used, otherwise secure code could be poisoned with external code.

Aim of this section is not being a substitution for the documentation of those specific libraries. Such a goal would be repeating old work and would not provide any useful data. Rather, the goal of this work is stressing out deficiencies of some highly popular C/C++ libraries. Sometimes, usage of a certain library is unavoidable; this work also gives information how to use possibly insecure libraries safely.

◊ Tools

Humanity owes to the tools for the advancement of civilization; tools make works easier and possible. This assertion is valid for the software engineering, as well. Engineers can use tools to build the product faster, more easily and more secure.

This thesis will provide information about the useful tools that helps making software more secure. We are unaware of any related work about this subject in academic environments.

2. Attacker

This chapter should be considered as an introduction to the subject. Furthermore, it gives background information to develop strategies in following chapters. This is important, because no defense strategy can be reliable without knowing the attacker. Additionally, motivations of enemies are defined to make designer aware that there can be different motivations and virtually nobody is safe without effective countermeasures.

Information about attackers and their motivations are generally not given in works related to secure software development. This thesis has this rarely seen attribute and makes it therefore unique if other topics are taken account, too. However, [27] is a book that is devoted entirely to the physiology of a hacker, and therefore it is a recommended reading for individuals, who are in the interest of knowing their attackers better. Coverage in this chapter will therefore be limited to background information level.

2.1. Attacker

Internet brings computers closer, virtually next to each other. Every computer has neighbors, both good ones and bad ones. It is impossible to know who next-door neighbor is; attackers can be anyone from fourteen-year-old teenager using hacking tools he found on the web to the government with all of vast funds and experts.

Software designers should consider attacker as an anonymous entity with

- Full knowledge of internals of designed software (since this information can be achieved with reverse engineering),
- Strong (however, maybe yet unknown) motivation,
- Unlimited desire and patience for breaking into software (since this is correct way of thinking, otherwise it would be a very bad and costly underestimate), and

- Vast amount (finite, yet incomprehensibly huge amount) of computing power.

2.2. Motivation of the Attacker

Attackers can have different motivations usually leading to monetary (like making money) or social gains (like being famous or developing self-respect). Knowledge about possible motivations will help developers to understand threats presented in a networked environment.

2.2.1. Monetary Gains

2.2.1.1. Stealing Money

Attacker may have discovered a way to transfer funds from a bank directly to an account under his control. To accomplish this, he tries to break in the software and force it to do what it would not normally do. Although this motivation is most well known motivation among public, it is actually not so popular among hackers, because of extreme difficulties and risk involved.

2.2.1.2. Blackmail

Attacker may have discovered a vulnerability severe enough that might draw interest from other attackers. Attacker can blackmail company representatives with releasing that specific sensitive information to public domain potentially causing more attacks and bad press. He can send some stolen information as a proof of concept.

2.2.1.3. Ransom

Attacker may have succeeded stealing information from a company, but information itself may not be valuable to attacker. However, this information will be probably valuable for someone else and company may want to give ransom money to stop the attacker from releasing that information.

2.2.1.4. As a Job

Attacker may be doing attacks as a part of his paid job. For instance, he might be paid by a company to discover vulnerabilities in competitor products with a hope of

causing bad press, reducing sales and eventually increasing its market share and profits. Another example can be software security analysts working for government.

2.2.1.5. Finding a Job

Attacker may be in hope to be noticed by one of computer security companies and being offered a job with high salary.

2.2.1.6. Stealing CPU Cycles

There may be a high prized contest, which requires a high amount of computing resources. Attacker can write a worm to sneak into millions of computers and make them to compute what he wants. Additionally, that computing power can also be used to perform a more focused attack to a specific company, possibly with one of other motivations described in this section in mind. This can be base of a denial of service attack.

2.2.2. Social Gains

2.2.2.1. Gaining Self Respect

Attacker may feel himself better or find satisfaction that he cannot find socially elsewhere by proving his intelligence and talents to himself with an accomplishment of successful attack.

2.2.2.2. Giving Message

Attacker may have a message to declare the world and can seek a path, which involves breaking into computers and displaying a message. Contents of message can be anything and may range from declaration of love to a loved one to the extent of political issues. Besides just showing the message, attacker can decide to do actual harm to make message more unforgettable and noticed, better yet mentioned about in the evening news on TV.

2.2.2.3. Being Famous

Attacker may be in desire gaining social acceptance in hacker communities with successful attacks.

2.2.3. Other Gains

2.2.3.1. Military and Armed Forces

Military would definitely want to decipher tactical and strategically information from competitor country forces during wartime and peacetime.

2.2.3.2. Intelligence Services

Intelligence services will definitely try to discover more information by learning secret data. Since cryptographic algorithms are generally very hard to break because of their well studies theoretical background, it will most likely much easier to break in computer systems and access plain-text information directly.

2.2.3.3. Police and Armed Forces

Police may want to access secret data for evidence, proof and tracing. If suspect is using computer for anything related to its crime, it may be worth trying to break into software since it can be easier, more subtle and safer than breaking into house physically.

3. Attacks

Understanding different kinds of attacks is required to be able to write vulnerability free code and develop strategies. Different attack methods are different instruments of enemies.

“Server Side Attacks” section describes attacks for hosts providing a service in a hostile environment. Term “server” as used here does not necessarily mean big machines with multi CPU’s in cooled system rooms, desktop computers may also serve services as well, like peer to peer networking or personal web sites.

“Client Side Attacks” section describes attacks for hosts consuming some sort of service from a hostile server or from a legitimate server used as a leverage to redirect client to a hostile server. When an administrator of a server starts browsing a popular site, the machine becomes a client machine and thus vulnerable to client side attacks.

This chapter also gives examples of actual attacks. Attacks are chosen among others with the criteria of being widely known and having high damage.

3.1. Server Side Attacks

3.1.1. Introduction

Since servers are shared among many people, even one successful attack to a single server causes broader damage to public and more gain to attacker. Therefore, they are generally more popular and better known. Attackers usually prefer directed or common attacks to servers rather than attacking to clients individually, because of possible higher-profit outcome.

3.1.2. Sample Attacks

Sample attacks from the near past are listed below as examples:

◊ Reportedly, nineteen-year-old Russian called Maxim stolen credit card, address information and other private data of some 300.000 customers, and wanted $100.000 ransom [2].

◊ Code Red infected servers running Microsoft IIS server on Windows 2000. Cost is estimated over $2 billion. It clogged network bandwidth, allowed attackers to take control of servers, and caused information theft. [3]

◊ MyDoom [4] worm infected more than one million computers worldwide. It was responsible 20% of email messages sent globally at that time (Jan 2004). It has slowed down internet more than 50 percent and made DoS attacks to some companies including Microsoft, Google, AltaVista, Lycos and SCO, causing SCO to change its domain name. Estimates are that MyDoom has caused $40 billion in economic damage.

◊ Two buffer overflows (one heap and the other stack) in name resolution service of Microsoft SQL Server 2000 caused security vulnerabilities. Those vulnerabilities are exploited by Slammer worm and at least 22.000 servers are affected by it. [19]

◊ Blaster worm [20] [21] took advantage of buffer overflow in DCOM remote activation implemented in RPCSS.DLL in all major versions of Microsoft Windows including 2000, XP and Server 2003 allowing remote attacker to run arbitrary code in the context “Local System” account. That account is one of the most powerful accounts in MS Windows OS, it could do possibly anything that an administrator could do on the system console.

3.1.3. Denial of Service (DoS) Attacks

Denial of service attacks are designed to interrupt service provided by servers connected to Internet. It has three major mechanisms.

◊ Attacker Consumes Network Resources

Attacker sends extensive amount of network packets to the server where packet contents are not important and just consume bandwidth. Sending hundreds of thousands of PING packets can be an example for such an attack. This kind of attack must be stopped on network devices like intelligent routers, firewalls, or intrusion

detection systems (IDS); or by ISP's. They are not security vulnerabilities of software.

◊ Attacker Uses Server Resources

Attacker sends low cost high impact packets to the server. TCP servers can be attacked by sending large numbers of TCP-SYN packets (only 40 bytes with IP header) each causing server to prepare for a TCP connection and allocate resources. For UDP servers, attacker can send a large number of requests for a time consuming service (like authentication). These kinds of attacks can be detected by intrusion detection systems (IDS) and may be prevented with proper reconfiguration of network devices. However, application programmer can take countermeasures to reduce the chance of successful attacks.

◊ Attacker Crashes Server

Attacker manages to discover a vulnerability of the server application. He sends a specially crafted packet to the system, either causing server to allocate extensive amount of resources finally bringing it down or crashing (maybe because of a general protection error caused by a buffer overflow, which is possibly caused by an integer overflow) instantly. This kind of attack is almost impossible to detect by IDSs, at least before updating detection engine on the IDS with the signature of that specific attack. Application programmers are responsible and accountable for attacks resulting in server crash.

DoS attacks can be used as a leverage for attacks that are more sophisticated; for example by keeping IDS busy and hiding password-guessing attacks among other packets.

DoS attacks may be performed distributed by multiple hosts; this is then called **Distributed DoS**, or DDoS. This type makes it even harder to detect attacker and to prevent attacks. Attacker first writes a virus and infects computers of normally legitimate users. After a sufficient amount of time to spread around, it triggers attack and thousands of hosts globally attacks to a specific server. While DoS attack can stopped easily by modifying IP access control lists on the routers or firewalls, preventing DDoS with that method is impossible because of high amplitude of hostile connections.

3.1.4. Remote Code Execution

This attack is most frightening type of attacks. It allows attacker to gain complete user rights as the user context that the infected program is running in. If the user happens to be administrator, attacker practically owns remote computer and can make it to do everything he or she wants.

This type of attack uses buffer overruns (simple buffer overruns, buffer overruns caused by integer overflows or internal state confusion) and gets more effective as the user context of the attacked program gets more privileged.

Worms are generally used to make spreading more effective. A worm is a malicious program that enters into a system from a security hole (like the ones caused by different flavors of buffer overruns). After infection, it generally tries to spread itself to other systems by probing network and sending specially crafted network packages (generally same packet is used to sneak into other systems).

3.1.5. Server Hijacking

This form of attack is generally performed locally by a malicious administrator. Legitimate server application is replaced with similar looking malicious one in the hope of collecting sensitive user information. In some forms, legitimate server continues to run along with malicious software (malware) and report its status as okay.

The malware does not have to be full-blown implementation of legitimate server application; generally, only front end is implemented. After users reveal their account information, server responds with a report of some internal server error advising to try again a few minutes later, rather than showing incorrect information and making users to suspect.

Most popular methods are completely replacing application; installing malformed one with TCP binding hijacking; redirecting user requests with network equipment or with configuration in server (such as in TACACS+ “Follow” command).

3.1.6. SQL Poisoning

SQL poisoning is an attack that is performed by supplying server application input parameters, which actually conceal harmful SQL commands. An application lacking

proper input validation will use those parameters while building SQL query, which turns into a harmful SQL command. A very simple example can be as follows: Assume that developer checks authenticity of the users with a SQL statement that is constructed with following C code:

```
1       sprintf(
2               szFinalQueryString,
3               "select count(*) from accounts where"
4               " username='%1' and"
5               " password='%2'",
6               szInputParameterUserName,
7               szInputParameterPassword;
```

Figure 3.1: Sample code for SQL poisoning

A user supplying username as "*some string' or 1=1* --" and password as any "*some string*" will gain access to the server no matter if there is an account or not. After construction, resulting string will be

```
8       select count(*) from accounts where username='some_string' or
1=1
9       --' or 1=1; #and password='some_string'
```

Figure 3.2: Resulting SQL command of sample SQL poisoning code

As seen above, Line 9 is completely an SQL comment, since "--" is SQL comment delimiter.

More harmful attacks can drop tables, delete databases or, in the worst case reveal user information. There is a high quality previous work in this area. Therefore, preventions of SQL poisoning will not be discussed in this thesis. [27], for instance has a good deal of information about Microsoft Corporation SQL Server security, and it covers SQL poisoning, too.

3.2. Client Side Attacks

3.2.1. Introduction

Client computers are generally administrated by people who are less knowledgeable in computing systems administration than the administrators of servers. Therefore, client side attacks, differently than server side attacks, generally depend on lack of knowledge of users.

3.2.2. Sample Attacks

Below can be found sample successful attacks from near past.

◊ Melissa Worm infected computers with malicious Microsoft Word documents in some versions of Microsoft Word application. It has impersonated users and sent their private data as attachments to the contacts extracted from their computer. It also deleted some critical system files. [5]

◊ "I love you" virus, appearing in May 2000, sent mail messages to every contact extracted from infected computer with the subject line "ILOVEYOU" and a VBScript attachment. It caused email traffic blockage and an estimated economical damage of $10 billion.

◊ SoBig worm [22], released in August 2003, is a trojan which is spread to contacts extracted from infected computers via attachment in an email and caused high volume of malicious traffic in Internet, blocking legitimate traffic. It has caused a high amount of financial lose and inconvenience [23].

3.2.3. Trojan Horses

Abbreviated incorrectly to trojan, its name comes from historic Trojan Horse and designates a type of attacks where malicious software is buried into seemingly harmless useful software. Once the user is convinced to run the program, malicious part of the software becomes active and does its harm. They generally install a root kit to open a back door to the infected system, log key strokes possibly to learn passwords and other private information; all of them to impersonate user.

A "Root Kit" is a piece of software that runs in kernel mode and becomes part of operating system, which means that it becomes part of trusted computing base

(TCB). In theory, it is possible to create a root kit that is impossible to be detected or removed. It can trap, redirect and modify system calls and their return values, change scheduling and inter-process communication; in shorter words, everything that an operating system can. This can be considered as a perfect camouflage.

Systems can be protected in three ways against these attacks: First, there can be virus protection software that detect suspicious activity before it happens. New viruses can work around this. Second, all software can be digitally signed by the original manufacturer. If the contents of package are tampered with, signature will mismatch and operating system will detect. Attackers still can write their own software and even sign it. However, it is very difficult to convince a well-known root certificate authority to sign their certificates. Attacker can setup its own CA; but this will generate "Not trusted CA" warning. Nevertheless, this time, user might not understand what all of this jargon is and just choose running software. To mitigate this case, second, a more general strategy can be used: running the system as a non-administrator account. It will be harder to infect systems. Even if the system is infected, potential of the harm will be limited since the attack surface will be much smaller. For example, non-administrator accounts in Microsoft Windows cannot install kernel mode drivers, which makes root kit installation impossible (unless there is vulnerability in Windows OS itself, of course).

3.2.4. Viruses

Viruses are very similar to trojans, with one difference that they are designed to spread themselves and try to infect other computers aggressively.

3.2.5. Cross Side Scripting (XSS)

Cross side scripting is a form of attack that is performed by putting malicious scripts in a trustworthy context and deceiving people to run them. For example, an attacker can supply a comment with a client-side script buried inside to a blog site. Visitors of that site will run this script in the context of that site. Script can ask for username and password, or alternatively steal session cookie, and send those information back to the attacker.

3.2.6. Phising

Being a subcategory of social engineering, phising is requesting sensitive information from users by deceiving them as such requests come from legitimate representatives. Attackers send official looking mail messages to users and try to convince them to reveal their personal information by claiming that there is some problem with their account. The mail further says that by clicking a link in the mail and entering user info will solve that problem. Client following directions from those mails end up in hostile web sites those steals their password.

Some phising methods include URL spoofing like using complicated IP addresses which regular users will not understand its destination or using fake domain names similar to official ones as in http://www.hotmail-supersecure.com or http://www.hotmai1.com (please note numeric one "1" at the end). Even appearance can be forged to be same, for example, http://www.hotmail.com has Cyrillic letters such as "o" and "m" as it results in different domain name thanks to Unicode DNS system. Using details of Internet URI resolution (as in http://www.bankofamerica.com@www.hostile.com) is another method. Other phising methods use fake mail messages with malicious software as the attachment convincing clients that they are coming from trusted contacts.

Phishers exploit deception of human mind, not security vulnerabilities in software. However, software can be designed to protect users from deception by warning them against possible threats. This thesis does not cover countermeasures for phising attacks, as they are not directly related to code defects. Sound design principles must be followed to prevent phising from happening. Readers are urged to refer to [24] for preventing misuse and false security sense of two-way authentication mechanism. [25] presents US-CERT report and unfortunately, as of March 2005, there is a trend of 26% increase in phising attacks. [26] and [27] are good resources for more information about social engineering. Especially Chapter 10 in [27], "Social Engineers -How They Work and How to Stop Them", is a good introduction to the subject.

4. Requirement Analysis

4.1. Motivation

Basic principle in requirement analysis of a secure system is defining requirements precisely to ensure that only required features are added to the list, this ensures keeping attack surface small. For instance, if dynamic update from the network feature is not required, adding that feature increases opportunities for attackers unnecessarily.

Another very important analysis during this phase is security needs of the product. Who is the audience of this product? What are the security-usability trade-offs that can be made? These decisions play a very important role in the overall security of the system.

4.2. Previous Work

Waterfall methodology approach revealed that requirement analysis phase is researched very well since it is one of the main aspects of software engineering and many other engineering disciplines as well. We have nothing to add novel to this area of software development life cycle.

Researches are encouraged to investigate opportunities in requirement analysis phase, which allows programs to be safer in the meaning of their existence.

5. Design

5.1. Motivation

A good and secure design is key element of secure computing. A trustworthy computing system must be "**secure by design"**. Insecure designs are almost impossible to retrofit with security features later to make them completely secure; there will always be an attack, may to be yet discovered.

5.2. Previous work

Designing high quality software involves a very broad range of subjects. This thesis does not repeat rich previous in this vast area; rather it presents important subjects that are not previously worked on, at least in this context. Motivation on that subject and previous work are detailed at the beginning of each subject.

5.3. Tight Tunnel

We define tight tunnel as an execution path with minimal unexpected paths. Surprises are disastrous in software; therefore tight tunnel operation is crucial in software systems.

5.3.1. Motivation

Programmers must be very precise when ordering commands to computers, because computers do not have commonsense like people do. In normal life, scope and applicability of the commands and rules can be obvious. In the realm of computers, everything and anything must be set in order precisely to prevent possible gaps in the interpretation. Unfortunately, ensuring a tight tunnel for each possible execution path is a difficult task in software project. This can mostly be achieved in design phase and therefore it is handled in this chapter.

5.3.2. Previous Work

Steve McConnel's book [16] makes a great job in defining principles of good code development. Although that book has vast information in overall quality of design and development, it does not have information in the context that is presented here. [52] is another popular book which gives information about best practices in development. However, that book, too, lacks of information about design decisions of code structure. There is not academic article about this subject that we are aware of.

5.3.3. Concept

Code must be designed to flow in tightest tunnel possible. What is meant here is that code execution must be restricted with language and operating system features to the maximum point as much as possible. Examples, which are sorted from low to high level, are below:

- Variables should be declared
    - as const if they will not be modified later
    - appropriate in size, not larger or smaller than needed
    - as unsigned if signed operations are not needed (Counter are especially candidates for unsigned integers)
    - with minimum visibility to outside (usage of namespaces and public/private namespaces is recommended)
- Functions should be declared
    - with parameters that complies with principles of variable declarations and supports clear "in" and "out" parameters
    - as const, if they will not directly or indirectly (via non-const function calls) modify member variables later. Type casting or mutable declarations can be considered.
    - with minimum number of overloaded variations
    - as reusable and as generic as possible
    - with throw specification if possible

    - with minimum visibility to outside (usage of namespaces and public/private namespaces is recommended. Private / Protected difference should be respected and Private should be preferred over Protected to prevent derived class namespace bloat.)
    - with consistent error handling, using exceptions of some standard type for all error reporting is highly encouraged.
    - Functions that are not returning at all should be declared as `__declspec(noreturn)` (Microsoft Corporation C/C++ compiler) or `__attribute__((noreturn))` (GNU GCC)

- Classes should be declared
    - with minimum number of inheritances
    - with parameters that complies to variable declaration principles
    - with functions that complies to function declaration principles
    - with minimum number of constructors with maximum number of default parameters possible
    - by hiding unused constructors as privates to prevent copying etc.
    - with minimum number of friend functions possible
    - with minimum number of casting operators possible
    - with minimum visibility to outside (usage of namespaces and public/private namespaces is recommended)

Regarding these guidelines can prevent many of the bugs by detecting them at the compile time. Another note is that C++ compilers support these principles as a part of standard. Preferring C++ to C can be rewarding even if no object-oriented design is targeted.

5.3.4. Advanced Topics

Tight tunnel is not just variable, function and class declarations, the concept involve more. Data flow, for instance must be also in a tight tunnel. This means that designer should design interfaces in a way that all of them use same type of data (only meters,

not centimeters or kilometers, for instance). This allows data to remain in the same meaning throughout the execution process.

Another important aspect is that function names and variable names must be declared and used consistently so that programmer mindset stays tuned to only one kind of standard. Pre-pending function names or grouping them in namespaces is therefore a good idea. It keeps less and clearer choices to the programmer to select from, which of course results in tighter path for execution.

5.3.5. Examples

For instance, if these guidelines would be followed, following bug from latest Linux kernel at the time of writing would be discovered much earlier than it was, because compiler would have warned against signed-unsigned mismatch [51]:

> *Date: Wed Aug 3 18:43:22 2005 -0700*
> *[PATCH] sys_set_mempolicy() doesn't check if mode < 0*
> *A kernel BUG () is triggered by a call to set_mempolicy () with a negative first argument. This is because the mode is declared as an int, and the validity check does not check < 0 values*

Similarly, following bug could have been prevented if GCC dictates tight tunnel principle better [51]. Explanation of the bug is inside the cited text:

> *Date: Tue Jun 28 20:45:06 2005 -0700 Variable "c" was declared as an unsigned int, but used in:*
> *[PATCH] coverity: i386: build.c: negative return to unsigned fix*
> *125 for (i=0 ; (c=read(fd, buf, sizeof(buf)))>0 ; i+=c)*
> *126 if (write(1, buf, c) != c)*
> *127 die("Write call failed");*
> *(akpm: read() can return -1. If it does, we fill the disk up with garbage).*

Another tight tunnel problem with GCC is following comments from same thread [51]:

> *Date: Thu Aug 18 14:40:00 2005 -0700*
> *[IA64] remove unused function __ia64_get_io_port_base*
> *Not only was this unused, but its somewhat eccentric declaration of "static inline const unsigned long" gives gcc4 heartburn.*

These and other examples imply that tight tunnel principle is not in common practice in the degree as it should be. Potential cause for this can be lack of knowledge among developers.

Additional disrespect to tight tunnel from same change log is seen below. GCC should have warned comparison between signed and unsigned variables.

> *Date: Thu Aug 4 19:52:03 2005 -0700*
> *[PATCH] __vm_enough_memory() signedness fix*
> *...*
> *We hunted down the problem to this:*
> *The deferred update mecanism used in vm_acct_memory(), on a SMP system, allows the vm_committed_space counter to have a negative value. This should not be a problem since this counter is known to be inaccurate.*
>
> *But in __vm_enough_memory() this counter is compared to the `allowed' variable, which is an unsigned long. This comparison is broken since it will consider the negative values of vm_committed_space to be huge positive values, resulting in a memory allocation failure.*

Tight tunnel is not only useful in security but also in optimizations. If compiler precisely knows what is exactly intended to be done, then it can optimize code accordingly. Below is an example:

> *Date: Tue Jun 21 17:14:55 2005 -0700*
> *[PATCH] __read_page_state(): pass unsigned long instead of unsigned*
> *By making the offset argument of __read_page_state an unsigned long instead of unsigned, we can avoid forcing the compiler to sign extend a usually constant argument. This saves 1 instruction on x86-64.*

5.4. Design Patterns

This section analyses selected design patterns from a security point of view. Design patterns are selected from famous pattern catalog “Design Patterns” of Erich Gamma et al [14]. Selection of patterns made by their popularity and whether they have an important aspect of security or not.

5.4.1. Motivation

Design patterns are used all over the world for different software projects, since it makes understanding of the project design easier and universal. Moreover, if used correctly, design patterns result in more manageable and easier to implement design. As people use words in their sentences to describe something, patterns help describing internals of software design.

Examining design patterns deeply reveal that they have different security wise aspects, which are very important to build trustworthy applications. Some of the patterns add inherited robust design that results in more secure code, while others

pose some threats that should be mitigated in order to use that pattern safely. Since design patterns are in wide use, describing those aspects are very important.

It is important to note that this work not only discusses weaknesses of existing patterns, but also it extends their use where applicable.

5.4.2. Previous Work

There are numerous articles and books about design patterns and about best practices to use them. Unfortunately, those resources fall short to define security aspect of the patterns. There are articles that define brand new patterns for security related applications, however this does not help using old and more commonly used patterns in a secure fashion. At the time of this writing, this work is the only one about this subject.

5.4.3. Creational Patterns

5.4.3.1. Prototype

This pattern is very useful to reduce class count, thus complexity. Moreover, it helps code reuse, which is a good trait for secure software since it decreases the number of lines where a code defect can be introduced into the code. Code reuse helps furthermore by increasing test coverage. Therefore, this pattern is highly recommended for class hierarchies with similar classes.

Major concern is that abstract classes only define interfaces, and interface level agreement does not guarantee implementation level compatibility. In this pattern, there is one interface pointer, which can actually point to one of multiple concrete classes that are unknown at the runtime. If implementations of those concrete classes are incompatible, bad consequences can scale up to buffer overruns. To prevent this from happening, interfaces must be designed very clearly, with only required parameters in the same meanings (please visit previous section for further discussion about tight tunnel principles). Although it can be considered paranoiac, minimizing (or eliminating if possible) usage of pointers, especially the ones that are passed to other classes, is safer way to go.

```
10      //************************************************
11      //************************************************
12      class String {
13      private:
14            char * pStr;
15      public:
16            char * Set(char *pStr) = 0;
17            char * Get() = 0;
18            int GetLength() = 0;
19            ...
20      };
21
22      //************************************************
23      //************************************************
24      class UppercaseString {
25            char * Set(char *pStr) { }
26            char * Get() { }
27
28            int GetLength() {
29                  int iLength;
30                  char * pItr = pStr;
31                  for (
32                        iLength = 0;
33                        *pItr != 0;
34                        ++iLength, ++pItr);
35                  return iLength; }
36      };
37
38      //************************************************
39      //************************************************
40      class LowercaseString {
41            char * Set(char *pStr) { }
```

```
42              char * Get() { }
43
44              int GetLength() {
45                      int iLength;
46                      char * pItr = pStr;
47                      for (
48                              iLength = 0;
49                              *pItr != (char)-1;
50                              ++iLength, ++pItr);
51                      return iLength; }
52      }
```

Figure 5.1: Prototype Pattern Misuse Code

Above is a demonstration of mismatched prototype implementation. Example is very simple and provided only as proof of concept: One string represents end of string with (-1) while other represents with NUL. In real world scenarios, classes will be much more complicated and much harder to test.

5.4.3.2. Factory Method

This popular pattern (that is used by COM feature of Microsoft Windows operating systems.) is a variation of prototype pattern. Same concerns are also applicable to this one. This pattern has some advantages of its own, as described next paragraph.

Normally, classes are configured for each user and than attached to the user context. This approach brings state and configuration data to classes, which is not very desirable. If there is a small amount of user types, there can be several subclasses. A factory method can create a custom class according to user type. These “hardwired” concrete classes will be playing one well-defined role, thus making implementation easier (and therefore safer).

For an example, please see following code. Let us assume that in an application, there are two types of users: “NormalUser” and “SuperUser”. Normal users can read the files, where super users can also write them. A straightforward approach could be having a Boolean member variable, which holds user type. Then, write function call

can check for this variable and perform appropriate operation. However, this approach is not good. First, function implementations get more complex. Second, keeping state information can be difficult and error prone. Finally, a stack smashing attack can easily change this member variable value and can elevate itself to a more privileged user account. A better approach is “hardwiring” functions to their users. This approach scales better with higher number of functions.

Below is an example implementation of program with two user types. Since there can be a very high number of subclasses, designer may want to incorporate other patterns to make development a bit easier by allowing code reuse among subclasses.

5.4.3.3. Abstract Factory

This pattern is very similar to previous pattern; it can be considered as a classed version of factory method. Concerns and advantages are practically the same.

```
53      //*************************************************
54      //*************************************************
55      class User {
56      public:
57              void ReadFile(...) {
58                      System->ReadFile(...);
59              void WriteFile(...) = 0;
60      };
61
62      //*************************************************
63      //*************************************************
64      class NormalUser : public User {
65      private:
66              ...
67      public:
68              void TruncateFile(...) {
69                      throw “Access Denied!”; }
70              Void WriteFile(...) {
```

```
71                  throw "Access Denied!"; }
72      };
73
74      //*************************************************
75      //*************************************************
76      class SuperUser : public User {
77      private:
78              ...
79      public:
80              void TruncateFile(...) {
81                      System->TruncateFile(...); }
82              Void WriteFile(...) {
83                      System->WriteFile(...); }
84      };
85
86      User * NewUser(bool fIsSuper) {
87              return (fIsSuper ? new SuperUser : new NormalUser); }
```

Figure 5.2: Factory Method Usage Example

5.4.4. Structural Patterns

5.4.4.1. Adapter

This pattern has an usage that is not mentioned in "Design Patterns". Besides other usage areas, adapters can be used when accessing insecure legacy functionality. Assume that there is a module, which provides a useful functionality that is time and/or money consuming to re-implement. However, it is designed in an insecure fashion (most probably because it is old and implemented before security awareness). Directly using that functionality can result in insecure application. If there is an adapter class, though, it can make all access checks, verification, authentication, authorization, accounting and all other types of filtering before passing those parameters to insecure parts. Moreover, it can use overwrite protected

heap memory (piece of memory that ends at the boundary of write denied page.) for "OUT" parameters.

Adapters can also be used as wrappers around old style C functions (most notably C runtime library) that do not have parameters to pass buffer sizes. This type of class would be more likely to be a wrapper class, though.

There might be consequences that adapter classes modify their input parameters. This is normally undesired because it can result in incompatibilities among classes. Adaptors can create their own copies, but this is undesirable because of performance reasons. There must be a good design decision on how much adapting and manipulation of an adapter is allowed to do. Readers are urged to refer to discussions in Design Patterns [14]. Another useful discussion can be found on Chapter 7: Field Notes of [29].

5.4.4.2. Bridge

Basic structure can be summarized as below. Essentially, classes A, B, C are using an abstract class bridge, which points to one of multiple implementation options. Classes A, B, C can be inherited from "Class Bridge", too.

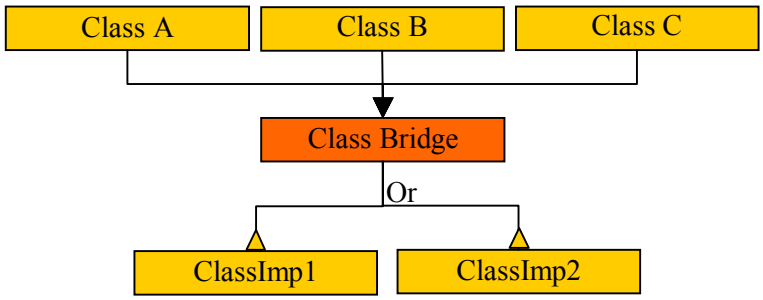


Figure 5.3: Bridge Pattern Structure

As discussed very well and detailed on [14], this pattern should be used where more than one infrastructure is going to be used. (Different infrastructures are encapsulated in ClassImp1 and ClassImp2.) This encourages code reuse highly. What make this pattern interesting from security point of view are platform dependent data type sizes and OS specific API's. ClassImp1 and 2 can be implementing classes for different architectures or platforms. This is transparent to upper level classes and encapsulated in Class Bridge. This significantly increases code reuse and simplifies design.

Static code analyzer tools like the one mentioned on [19], [30] is used during verification phase while porting software among platforms. However, this approach is different than using this pattern. First, these tools are designed for legacy applications and for retrofitting them with new security features, whereas this pattern should be used when starting a new project that must support different platforms and architectures to begin with. Second, such tools normally modify source code; large code bases modified based on some automatic tool recommendations can be scary if there are subtle bugs in the tool. Moreover, added source code makes runtime performance slower, which can discourage usage of harder encryption algorithms (which are very CPU intensive) or just fall below specification document. For instance, [30] can add to execution time up to thirty times of normal time. Finally, and most importantly, these tools are mostly used for detection of machine limitations and overflows, where this pattern could be supporting different platforms. Furthermore, capabilities are much broader with this pattern in a manner that this pattern supports also proprietary assembly instructions or API calls.

5.4.4.3. Decorator

No matter how much designers plan for possible features of products, there may be always requests in the future, which the product is not designed to accommodate. Then, designers and developers (who are usually different people than the original team) must modify the product. Each modification can add regressions and dangerous bugs into the code.

Decorator pattern allows projects to expand, as they need to. Therefore, this pattern should be used for pieces of code, where designers want to leave room for future growth or they are not sure of future requirements. Usage of this pattern is an investment in “Maintenance” phase of waterfall method.

5.4.4.4. Façade

This is another very useful pattern to prevent complexity, which is number one enemy of secure software. If a program depends on interfaces that are overly detailed and unnecessarily complex, designers can settle on a higher-level simpler interface and let this class translate these simple requests into series of complicated steps. At the extreme point, façade pattern can be compared to functions in programming languages; they both encapsulate a complicated process into a well-defined simple

interface. This interface then serves as universal interface to its users. A sample diagram of this pattern can be seen below.

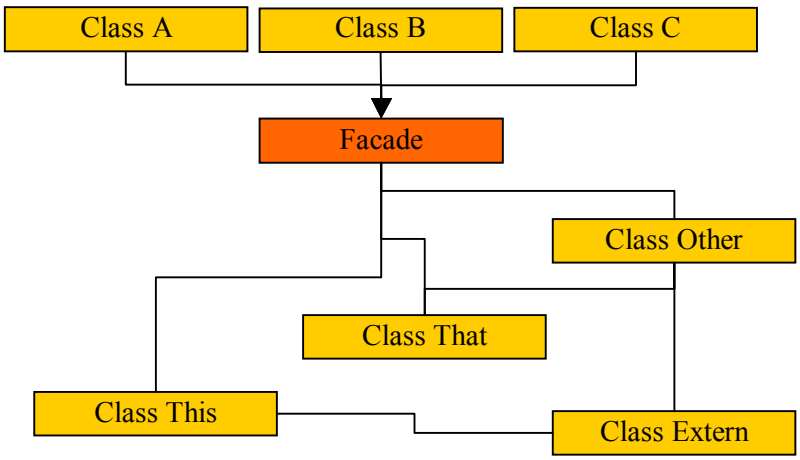


Figure 5.4: Facade Pattern Structure

This pattern is more useful if Façade class is used by more than one class since following complexity gets more difficult with more references.

5.4.4.5. Proxy

[14] Describes the difference between Proxy and Adapter pattern very well. As summary, adapter is used to change behavior of otherwise **incompatible class interfaces** whereas proxy is used to perform additional operations with the **same interface**. Still, from security point of view, these two patterns present similar traits and discussion about adapter is valid for this pattern as well. However, proxy has one more usage in that it enables additional AAA functionalities, namely Authentication, Authorization and Accounting.

AAA functionalities can be added easily with usage of Proxy: only thing to do is deriving proxy class from main class, overriding its member functions and adding required AAA functionality. This approach will not break clients (provided that they are referring to the main class over a pointer) and will not require any modification to the main class, which prevents regressions.

```
88      TimeClass gTime;
89      SystemClass gSystem;
90
91      //*************************************************
92      //*************************************************
93      class FileSystem {
94      private:
95              ...
96      public:
97              void CreateFile(...) { ... }
98              void DeleteFile(...) { ... }
99              void ReadFile(...) { ... }
100             void WriteFile(...) { ... }
101             ...
102     };
103
104     //*************************************************
105     //*************************************************
106     class FileSystemProxy : public FileSystem {
107     private:
108             LoggerClass _Logger;
109     public:
110             void CreateFile(...) {
111                     FileSystem::CreateFile(...);
112                     _Logger.LogCreation(
113
114                             gTime.GetCurrentTime(),
115                             gSystem.GetUser()) ); }
116     };
```

Figure 5.5: Proxy Pattern Example

A simple demonstration of retrofitted accounting functionality to an existing file system class can be found above.

5.5. Encryption

Encryption is a very sensitive and generally less understood part of design. Doing it right requires a very good theoretical background and deep knowledge of mathematics and number theory. It is very hard to tell if scrambled bytes really protect the data and from which kind of threats it protects. Can it resists dictionary attack, brute force attack, differential analysis, known plain text and cipher-text attacks and replay attacks?

This chapter gives information for developers who have responsibility of making good security design decisions. Practices mentioned here will make software more resilient to attacks.

5.5.1. Motivation

Encryption algorithms are after all mathematical calculations using mathematical traits of some mathematical operations. It is not wise to tamper with the algorithms in search for better ones. It is easy to make an algorithm less secure when it is not known what the effects are. At the other times, it is well possible not to reach desired security; though the designer thinks he did so, security traits might not have been changed at all. Using double encryption wrong way with a block algorithm can be an example.

A developer obliged to use an algorithm with shorter keys may try to double the key size by encrypting the message two times with different keys.

$$\textit{Cipher-text} = \textit{Encrypt with } Key_2 \textit{ (Encrypt with } Key_1 \textit{ (Message))}$$

However, according Schneier in [6], if the algorithm is group, then there is always a Key_3 such that

$$\textit{Cipher-text} = \textit{Encrypt with } Key_2 \textit{ (Encrypt with } Key_1 \textit{ (Message))} = \textit{Encrypt with } Key_3 \textit{ (Message)}$$

Thus rendering effective key length to half, namely unchanged. Designing security with false sense of having 2n bits key length despite the fact having only n bits key length would be catastrophic. Even if this is not the case, Merkle and Hellman developed a method called meet-in-the-middle (MITM) to reduce attempt count from

2^{2n} to 2^{n+1}. [31] is simple definition for MITM. [32] is the original article that discusses MITM.

There are triple key methods to increase key length and security. Here,

Cipher-text = Encrypt with Key_1 (Decrypt With Key_2 (Encrypt with Key_1 (Message)))

Effective key length is 2n bits. However, many crypto analysts have shown that there could be still vulnerabilities [6].

As seen here, cryptology is very sensitive to external modifications. It is easy to have a false sense of security while it is difficult to build a real one. This work focuses on usage of encryption algorithms. First, software designers do not design their algorithms; they rather use existing and well studied public algorithms. Focusing on quality algorithm development should be work of another article that aims advancement in bottom layers of cryptology. Second, even a perfect algorithm can be turned into a weak one by using it improperly. Third, with even stronger encryption algorithms, weakest link is being shifted to other areas. Now, traffic analysis is one of the emerging threats for secure communications.

As a summary, motivation of this section comes from idea that correct usage of crypto algorithms unleashes full potential and prevents weaknesses.

5.5.2. Previous Work

Cryptology, thanks to its mathematics nature, is one of the most studied and advanced sciences in computer technology. There have been numerous works on algorithm design and powerful algorithms since the beginning of 20^{th} century. However, there is a lack of connection between mathematical theory and its usage in software projects. This thesis aims to close this gap by providing information on cryptographic modes.

5.5.3. Background Information: Cipher Types

5.5.3.1. Stream Ciphers

Stream ciphers have key stream generators that output a temporary second key for next bit to be encrypted. Structure can be illustrated as follows:

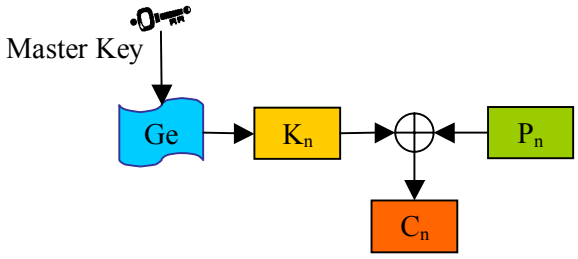


Figure 5.6: Reference model for stream ciphers

All security depends on the quality of generator. If it outputs predictable and routine stream of keys, it will be easy to break the cipher-text. On the other hand, it outputs seemingly random key bits with infinite period; attacker cannot break cipher-text without knowledge of key.

There are two kinds of stream ciphers: Self-synchronizing and synchronous.

◊ Synchronous

Synchronous stream ciphers have a key stream generator, which synchronizes itself with master key and then generates seemingly random key stream.

Since previous cipher-text output and plain text input does not influence key stream generation, bit flips do not cause error extension, only flipped bit will be affected in plain text; corresponding bit in plain text will be flipped, too. Designers must incorporate some sort of message integrity checking to their protocols, otherwise blindly flipping cipher-text bits will turn into a successful attack. Bit losses from or insertions to cipher-text will break alignment of decryption and will cause damaged decryption there.

Same key stream should not be used to encrypt different plain-text messages. Since key stream depends only on master key, using same master key will result in same key stream; there is no way to safely resynchronize key stream and encrypt even different plain-texts with that key stream. Applications should be designed in a way that avoids resynchronization of key stream. If that is unavoidable, new master key must be used to ensure security.

To decrypt n^{th} bit of cipher-text, key stream generator must be run to produce "n-1" bits of key stream (which will not be used). With larger cipher-text, this can be

severe performance penalty. Moreover, encryption cannot be parallelized because of this dependency. However, using different channels (for example channel number one encrypts every odd numbered block while channel number two encrypts every even numbered block) with separate keys and IV's (with inherent key management problem) can solve this.

◊ Self-synchronizing

Self-synchronizing stream ciphers takes key and previous "n" bits of cipher-text as input and generates key stream for next bit.

There is no need to start decryption from the beginning of cipher-text; only "n" bits history of cipher-text is enough to resynchronize key stream generator. This trait can also be security vulnerability. If a portion of previously captured cipher-text is replayed, synchronization will break; however, it will take only "n" bits of damaged decryption until resynchronization. If receiving end is not using any sort of integrity checking mechanism (like message authentication codes -MAC), damaged n bits cannot be detected and a replay-attack might be possible. Bit losses from or insertions to cipher-text will break alignment of decryption and will cause damaged decryption there after.

5.5.3.2. Block Ciphers

Block ciphers encrypt blocks of plain text by transforming each block with a mathematical function. Basic operation is as follows:

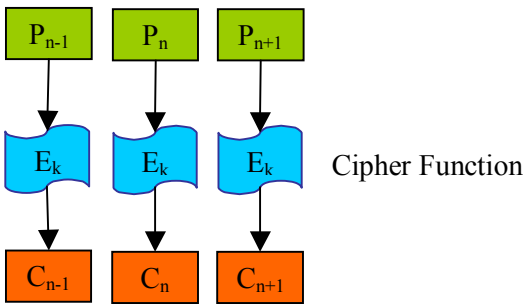


Figure 5.7: Block Cipher Operation

5.5.4. Encryption Modes

Algorithms define how to encrypt plain text data so that the content is kept secret. However, an eavesdropper can still infer useful information by just listening ongoing traffic like if there are any patterns (parts of cipher-text repeating itself) or there are any fixed header or trailers; he does not need to decrypt cipher-text to get this info. Besides, an eavesdropper may want to inject previously captured cipher-text data to replay transactions; plain usage of algorithm may not prevent this.

Encryption modes are customized usage of algorithms. It usually involves combining algorithms with some feedback or chain mechanism to gain security traits, which the algorithm cannot present naturally on its own. Usage of modes is a trade-off; they generally decrease encryption speed at varying levels, add complexity to code and effect fault tolerance. In this section, different modes are examined. Although effect to execution time is rarely a concern from security point of view, performance characteristics of different modes are given since it can affect usability of certain modes altogether.

5.5.4.1. Electronic Code Book Mode

Exactly one block of plain text is encrypted into exactly one block of cipher-text. It is even possible to consider each block as a message of its own. Encryption algorithm is illustrated below. Decryption is reverse flow of this process.

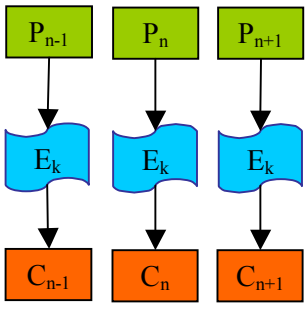


Figure 5.8: Reference model for ECB mode encryption

There is no relation between data blocks, this makes adding, removing, reordering and replaying of cipher-text blocks possible. Messages should be protected with message authentication code (MAC) algorithms, otherwise attackers can make harm without possession of key or knowledge of plain text.

Independence of blocks prevents error extension to other blocks in case of bit flips. Bit loses from or additions to cipher-text brakes synchronization in the block boundaries and results in defective decryption.

Since all blocks are independent from each other, blocks can be encrypted or decrypted in parallel. Random access to different parts of data is possible without additional decryption. There are no additional operations; encryption and decryption are as fast as underlying algorithm is.

5.5.4.2. Cipher Block Chaining Mode (CBC)

This mode uses previous cipher-texts as feedback to encryption to eliminate block independency. Each block of plain text is XOR'ed with cipher-text of previous block, and encrypted into a cipher-text block. Encryption algorithm is illustrated below. Decryption is the reverse of this process.

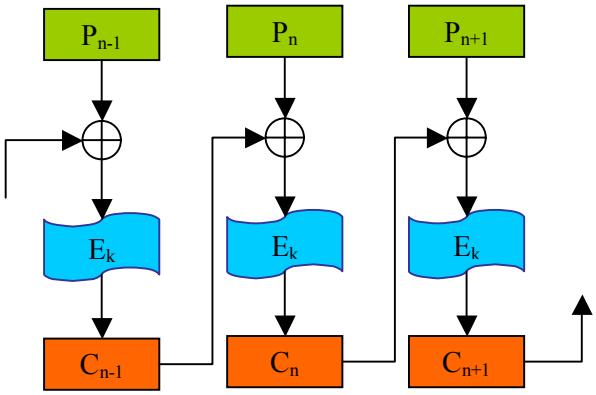


Figure 5.9: Reference model for CBC mode encryption

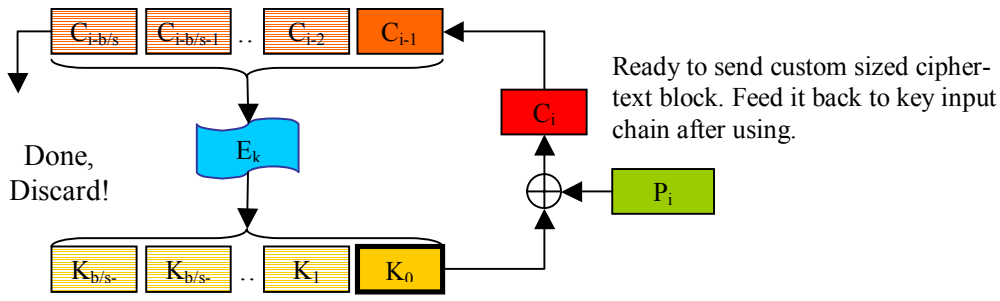
Each block is dependent to previous block, if previous block is different, same plain text block will result in different cipher-text blocks because each plain text is XOR'ed with different data. Follow of this dependency chain brings to conclusion that if first block is different, then all subsequent cipher-text will be different even if the plain text is same. Initialization vectors (IV) are used as initial feedback value to give uniqueness to first block. Random data as first block distinguishes cipher-texts of plain text. IV's does not need to be secret, after all each cipher-text used as feedback is an IV for later blocks. However, it must be unique and random.

Bit flips will result damage in its block and one block after it; two-block total will be damaged unrecoverable. Later block will still be good for decryption. Bit loses from or additions to cipher-text break synchronization of block boundaries and results in defective decryption.

Parallel processing is not possible since blocks are dependent to each other. However, using different channels (for example channel 1 encrypts every odd numbered block while channel to encrypts every even numbered block) with separate keys and IV's (with inherent key management problem) can solve this. Random access is not possible, either. To prevent starting from the beginning each time to decrypt data from random blocks, interim plain texts can be cached and that position can be used to start decryption instead. Caching plain-text data, however, is very hard to accomplish securely and can lead severe vulnerabilities. Additional problem is that extra XOR operations add performance overhead to process.

5.5.4.3. Cipher Feedback Mode (CFM)

Unlike block ciphers' fixed size (rather large blocks), CFM allows custom sized blocks to be encrypted with any block cipher. Operation is very similar to CBC; varying block size brings additional complexity, though.



B: Size of fixed sized blocks of underlying block cipher algorithm.
S: Size of custom sized blocks.
K: Custom sized key blocks
C: Custom sized cipher-text

Figure 5.10: Reference model for CFM mode encryption

Like CBC mode, usage of IV is required to ensure similar plain-text messages to be rendered into different and seemingly unrelated cipher-text messages. As will be described shortly, if attacker can flip bits in cipher-text, it will result in one partially

damaged custom sized plain text ($100 might turn into $228 with flip of bit 8) and "b/s" completely damaged custom sized plain texts. If application does not (or cannot) check for completely damaged plain text, it must use MAC to prevent tampering with integrity of packets.

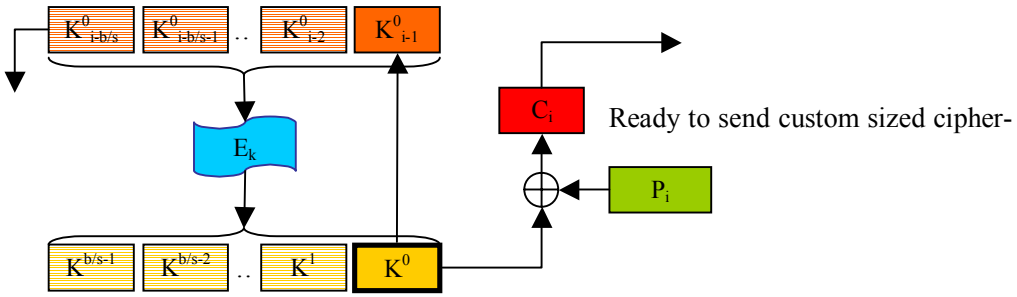
Bit flips damage bits of plain text that are in the same positions and plain text will be partially damaged. After then, corrupt cipher-text will be fed back to key register and it will cause completely damaged plain-text results until it is shifted out from the register, which takes "b/s" steps. Bit loses from or additions to cipher-text break synchronization of block boundaries and results in defective decryption.

Performance characteristics are similar to CBC mode. This mode has an additional benefit: Custom sized blocks can enable sending shorter network packets without the need of padding to the correct block size, which can be as much as 16 bytes.

This mode is an example of self-synchronizing stream cipher; only things needed to synchronize are "b/s" custom sized cipher-text blocks. It has traits of self-synchronizing stream ciphers mentioned earlier.

5.5.4.4. Output-Feedback Mode (OFM)

This mode is very similar to CFM, only difference is that key stream bits are directly used as feedback to feedback-chain.



B: Size of fixed sized blocks of underlying block cipher algorithm.
S: Size of custom sized blocks.
K: Custom sized key blocks
C: Custom sized cipher-text

Figure 5.11: Reference model for OFM mode encryption

This mode has similar characteristics like CFM. One advantage, however, is that there is no error extension. This can be also disadvantage: Bit flips will have impact

only on corresponding bit. In the absence of integrity checking, an attacker can toggle bits and modify plain text without knowing it. Bit loses from or additions to cipher-text brakes synchronization of block boundaries and results in defective decryption.

This mode is an example of synchronous stream cipher. Synchronization for the "n^{th}" bit needs generating "n-1" bits of key stream data. It has traits of synchronous stream ciphers mentioned earlier.

5.5.4.5. Traffic Analysis

Analysis of encrypted traffic is called Traffic Analysis and it aims to gather as much information as possible by analyzing encrypted traffic without any knowledge of plain-text data. This analysis can reveal message source, destination, length, time, frequency and match of this information with real life events like visits, meetings, working hours etc. Designers should decide whether analysis of their traffic is sensitive or not. Although lower layers of OSI can provide link-to-link encryption, (thus robustness against traffic analysis) higher levels usually provide only end-to-end encryption (thus leaking routing –L3 and transport –L4 information).

Choosing correct encryption mode with correct key management solves this problem. ECB mode is vulnerable to traffic analysis. CBC mode is secure against it. CFM and OFM modes are secure against traffic analysis, but reuse of key stream must be avoided.

5.5.4.6. Conclusion

Although different modes solve different problems, they all solve message secrecy related problems. Integrity and authentication must be provided by other means like MAC or digital signatures.

5.5.5. Paging of Memory to Disk

Although the application uses only dynamic memory and does not store any sensitive data in disk, virtual memory manager can decide to page out any data in memory, possibly including sensitive data. Long after the program exit, the data can reside in disk without any encryption. Thus, developers should use non-page able memory for sensitive information if analysis of disk by attackers is possible.

Allocating physical pages is possible in Windows Platform. AllocateUserPhysicalPages API call allocate requested amount of physical pages (not necessarily contiguous) which can be mapped to (contiguous) virtual addresses with MapUserPhysicalPages.

There are two problems with these functions: first, they allocate physical memory and remove that memory from use of virtual memory manager. Excessive direct use of physical memory will make system low on memory. The other problem is that "AllocateUserPhysicalPages" requires SE_LOCK_MEMORY_NAME privilege in the token, which is not default. Administrators must grant this privilege to the user of application from Local Security Policy console and developers must adjust process token to be able to successfully call "AllocateUserPhysicalPages".

A very good functional level example can be seen at [33].

5.6. Binary Design and Least Privileged Users (LUA)

5.6.1. Motivation

Programs run in user accounts in order to be authenticated, authorized and accounted for their actions. Assume that a program is running in a certain user context, every action in that program will have same user access rights, and so will the worms and remotely injected malicious code. Usually, not whole application necessarily requires all access privileges that some special parts may require. Modularizing programs into parts of their required user privileges and giving those parts "just enough" access rights will protect computers in case of faulty behavior and/or malicious activity. This practice is called running programs in "Least Privileged User" account.

One important thing to notice is that the use of "least privilege" does not necessarily mean really using least privileged user account in that system. It means that the user account that the program is running in has only minimum level of privileges that allows its execution, all other access rights are voluntarily given away.

There are several methods to enable LUA [15]. Although COM is widely used on MS Windows based applications, its use as a mechanism to enable LUA was not documented before. Extending capabilities of such a widely used technology is main motivation of this section.

5.6.2. Previous Work

[17] gives tremendous amount of information about DLLs in MS Windows and their OS level implementation. [34] gives information about creating DLLs and using them in applications. [35] and [36] gives information about how to enable COM authentication and authorization, which is required knowledge to implement ideas of this thesis. Unfortunately, COM is an old technology and its existence is well before security awareness. At the time that COM emerges, there have been some articles about its efficient usage. Later on, interest in COM (and other similar technologies like CORBA) is decreased with the introduction of later and advanced technologies. This hampered research on COM technology. Therefore, even after security pushes, researches did not go back and look at COM. Even security articles about LUA presented other approaches that had different advantages. However, author of this thesis decided to work on this technology because of the reasons mentioned on motivation section.

5.6.3. Background Information: DLLs

Dynamic Link Libraries have many advantages over static ones. First, as its name implies, it is possible to select desired library at the runtime. For example, different strategies with same interfaces can be distributed to different DLL's. At the runtime, executable can read an initialization file (or query registry in Microsoft Windows) and load desired library. Another advantage is that some DLL's do not get loaded until there is an explicit request to functionality in it. Those DLL's are called delay loaded DLL's and most of the DLL's can be specified as delay loadable during compile time of the library. Yet another good side with DLL's is that they are shared among different processes on the same machine. Unlike statically linked libraries, same instance of DLL is shared. For example, C runtime library is so widely used; statically linked programs will carry and load their copy of the library, thus wasting space and reducing likelihood of successful cache hits. Shared CRT could save space and since only one copy would reside in the memory, there is much less chance that it is swapped out.

In the security perspective, DLL's present some interesting traits: since DLL's help projects to be divided into smaller independent binary modules, it is possible to update modules individually. Assume that new security vulnerability is found in the

300 MB application. Users would have a very bad time to patch it from the Internet if the application is monolithic; users would have to download the whole 300 MB of data. This is very discouraging; users will try to combine downloads among different security updates, while in the meantime their systems will be left vulnerable. With DLL's in the scheme, it is possible to update DLL's that needs fix, which will be only a few hundred KB's usually.

However, on the downside, there are points to be aware while using DLL's. Since they are separate from executable binary, executable has no control on their authenticity. A malicious DLL presenting same interface can be named exactly as the old one and be replaced with it. Executable then will load this DLL, without knowing it is malicious. Operating system will run a digitally signed executable without user confirmation, however digital signatures are limited to executables. Executable will load malicious DLL and most probably will do things that are not intended at all. This problem can be resolved with DLL checksums and hashes.

5.6.4. Background Information: Privileges and Access Rights

Access rights are the rights that allow or deny selected set of users from performing certain operations. An operating system object can have an access control list, which defines who is allowed to do what and who is denied from doing what. This is similar in firewall or router access lists. Access rights are applied to single objects. Although objects (most notably file system entities) can inherit ACL's from their parents, this should not confuse the readers, objects still have their private ACL, only thing is setting ACL's on multiple files is done easier that way. For instance, file deletion, mutual exclusion object releasing, process terminations are all access right checked operations.

Privileges are global access rights and they do not apply to objects. For example, taking ownership of objects, kernel debugging, running in system context, allocating physical pages are special privileges that are granted on user bases. Some users have only a few of those privileges, more powerful ones (notably administrators or roots) have most or all of them. Since privileges have broader applicability, they are generally more powerful. For instance, a user account having the privilege of taking ownership of objects can read or write any object in that system, even if there is no specific ACL that grants performed operation.

5.6.5. COM Encapsulation

Component Object Model can be seen as advanced version of DLL's. They present same advantages and add other ones at their own. "DLL Hell" is not anymore an issue with COM; it has its own versioning scheme.

The interesting thing with COM from security perspective is that they can be loaded in process or out-of-process. If a COM binary is in process, it is pretty much the same as DLL's. Function calls will be handled in the same address space and privilege level as the host process. If a (for example "parser") function is compromised, then whole executable is compromised. Attacker will have exact security rights as the process, and every action the attacker takes will be accounted to the principle that created the process.

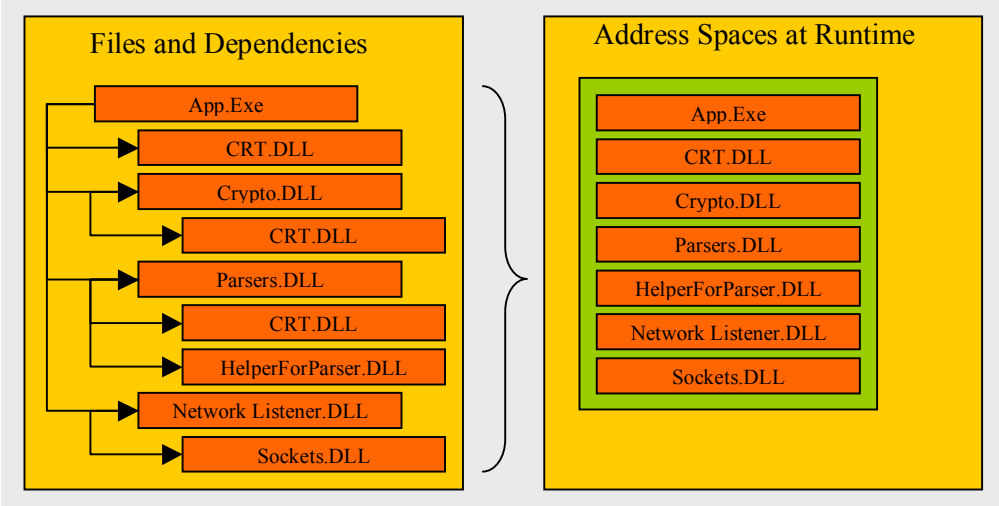


Figure 5.12: Address space with regular DLL usage.

Out-of-process servers, on the other hand, will have their own memory space and security context along with privileges. Since there is no direct address space mapping among executable and COM module, RPC mechanism is used to get them communicated properly. Every call will be marshaled and send via some transport protocol to the server. Replies from the server will be returned similarly. Please see below:

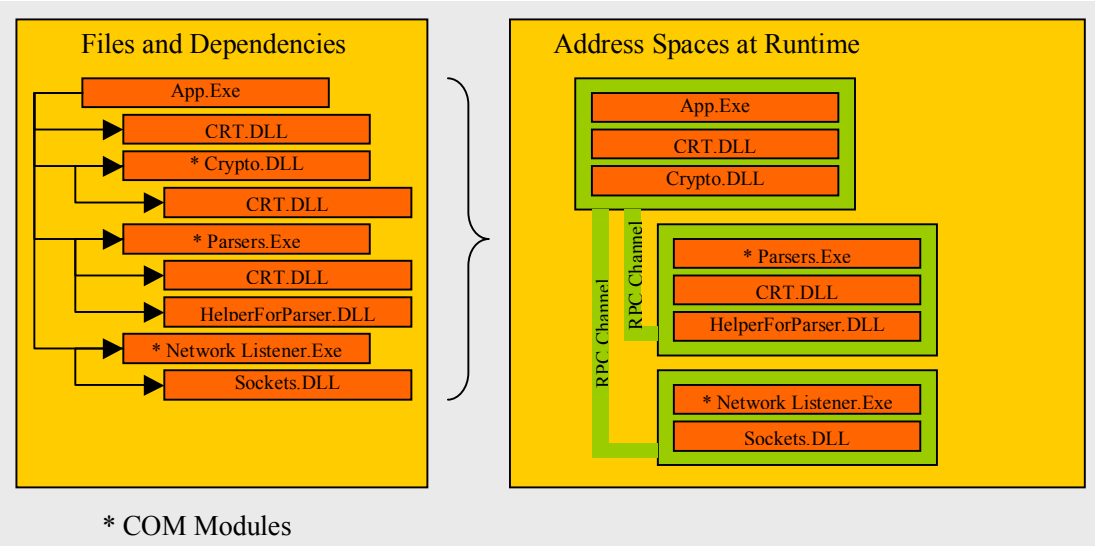


Figure 5.13: Address space with COM usage.

On the picture, “Crypto.DLL” represent an in-process COM server, thus it is loaded into the same address space as executable itself. “Parser.Exe” is a out-process COM module with some DLL dependencies of its own, thus it loaded into different address space with its DLL’s. Similarly, “NetworkListener.Exe” is another out-process COM module and it is loaded into another address space with its dependencies. Address space of executable does not have direct access to other spaces; it must use some inter process communication (IPC) mechanism. COM uses RPC as IPC. Each call to these different spaces is first marshaled, and then transferred with some transport protocol (LPC, TCP, UDP, etc.).

Beauty of this scheme is that different address spaces can have different security settings and privileges. If, for instance, “NetworkListener.Exe” process is compromised, attacker will be able to use only its access rights. In a security conscious system, this process would have almost no rights, which makes a successful attack almost useless.

Since RPC can use different transport protocols that are transparent to COM, different process spaces can reside in different physical machines. This is called Distributed COM, DCOM. This tremendous amount of flexibility allows architectures as below:

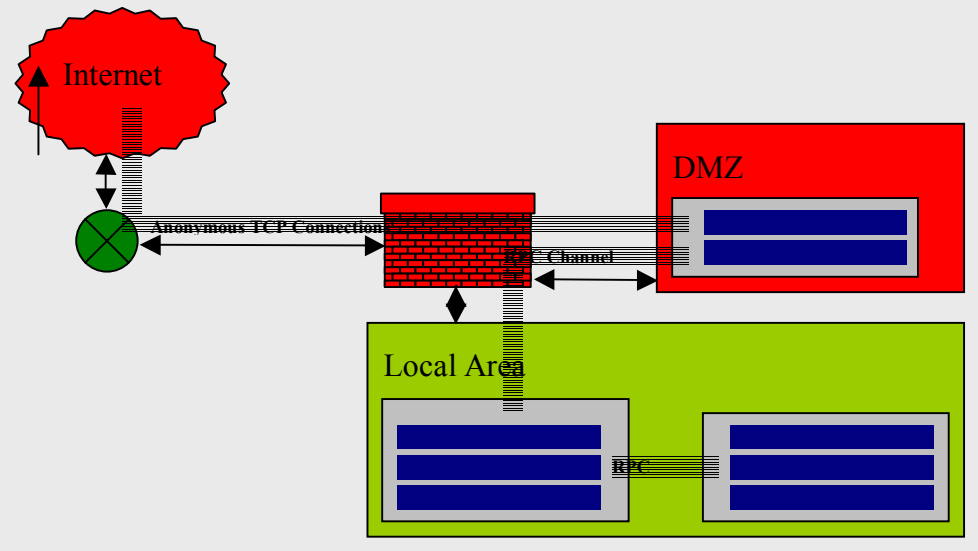


Figure 5.14: Sample architecture with out-of-process COM usage

"Figure 5.14: Sample architecture with out-of-process COM usage" presents a sample architecture, where computers of different roles reside in different and access controlled network segments.

5.6.6. COM+

COM+ COM with some advanced transaction services. These services make role based security and transaction management very easy and robust. DCOM developers should gain information about COM+, use integrated features rather than developing their custom code, and spend time on testing, improving, and hardening.

5.7. Threat Modeling

Threat modeling is required to understand and assess possible threats to the software. A good and sound design can be established only after such an analysis. Moreover, analysis of threats allows people at different position to understand possible threats and mitigations for these threats.

Since previous work is satisfying, this thesis will not detail threat modeling anymore. Especially [46] is one of the best general-purpose books about this subject.

6. Implementation

In this chapter, possible new improvements will be covered in the order of applicability: from one line to the whole program. All the examples are compiled and verified with Windows XP SP2 Build 2600.2158 and with Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86.

6.1. One Line Code Mistakes Catalog

6.1.1. Motivation

Software, obviously, consists of source code, which consists of lines. Theoretically, every code defect has a possibility to turn into security vulnerability. Having a catalog of example errors will increase developer awareness of possible caveats. Code defects are not necessarily inevitable; there are some methods to at least reduce the number of defects. This section aims providing those methods along with defects to help developers during implementation.

Main motivation of this section is showing that it is possible to reduce the number of defects by identifying them and providing countermeasures.

An objection can be that it is impossible to enumerate all of the defects those are one line. We agree with that criticism. However, it is possible to catalog families of code defects. There will be always some defects that are not mentioned in this thesis exactly; nevertheless, they will closely resemble the examples of this catalog and will be coming from same family.

Another objection can be that examples in this section are generally caused by careless developer; a good developer with high concentration would never make such mistakes and this section is therefore useless. We do not agree with this criticism. First, there will always be times that developers are not at their highest level of concentration. This can be caused by long working hours, stress of approaching deadlines, and magnitude of source code base, bad working environment and bad

tools. This thesis can help by spotting highest risk portions of code. Furthermore, this section can be used as code review checklist. Second, this section not only enumerates code defects, but also teaches countermeasures to prevent them from happening.

Yet another objection can be that examples of this section are not novel; one way or another, each example is still in some live code in some application. We agree with this criticism, most of the code defects in this section are not novel and they are not created specifically for this thesis. However, writing down code defects that is not occurred anywhere before is not a goal for this section, anyway. The goal of this section is cataloging and presenting of previously done code defects in order to show that they were evitable and they can be prevented in the future.

6.1.2. Previous Work

[16] is a great book on writing better code generally. Readers following the recommendation will benefit most likely. However, this work lacks of concentration of one-line code defects. There are numerous samples, but they are scattered throughout the book, reader has to read the book completely to access this information. Moreover, this thesis covers more code defects that are covered in that book. An older book, [37], gives an insight into C code defects. Nevertheless, the defects mentioned in that book are generally functional level defects, like not checking for NULL values or not using memory allocations correctly. [38] and [39] gives information about how to improve C++ usage. That book has one-line code improvements, which are actually very useful. However, they are improvements, not the defects. This thesis has another approach than that book.

6.1.3. Introduction

Every program consists of source codes, which ultimately consists of source lines. Any change to coding practices will improve source code quality dramatically.

Applicability and severity are provided to make extended usage of this thesis as a reference possible. Severity levels are enumerated as “Low,” “Medium” and “High.”

- Low: Only minor consequences are expected
- Medium: Attacker can use this as a leverage to a more severe attack.

- High: Attacker can inject code, escalate privileges, and execute remote code.

Applicability is categorized as “Limited” and “Broad.”

- Limited: Easy to find and fix. Introduced by
    - Only by lest frequently used features of programming languages, or,
    - Really distracted programmer, or,
    - Bad programming practices.
- Broad: Hard to find but easy to fix defects. Introduced by
    - Frequently used features of programming languages,
    - Minor distraction of programmer, even during following good programming practices.

Each title will have “(Sev: ###, App: ###)” decoration.

6.1.4. Integer Overflows (Sev: High, App: Broad)

6.1.4.1. Concept

All integral data types of C are kept in fixed finite size storage in memory or registers of CPU. For instance, integers can be defined to be 8, 16, 32, 64 or 128 bits wide and this storage cannot grow automatically; thus, maximum values of integral data types are predefined.

The problem with C is that if programmer tries to store a value with larger storage space requirements than current variable can provide, C assumes that this is intended by the programmer and silently trims overflowed part and fits remaining smaller value into existing storage. This happens without generating any kind of exception or error. Silently suppressing errors or converting variables implicitly mean bad practices of a language. Integers are of particular interest because they are generally used as counters or to hold lengths of buffers. Silently overflowing and presenting wrong length of an existing buffer or required length of a buffer results in security vulnerabilities caused by buffer overflows.

6.1.4.2. Background Information: Machine Representation

This paragraph describes machine representation of integers. Computers store integers in binary base-two format. An 8-bit integer means that computer will use eight binary digits to represent this particular integer value. Integer can be signed, or unsigned. Unsigned integers can only represent zero or positive values where signed integers can represent negative values below zero, too. Unsigned integers utilize every binary digit to represent the value. For example, 8-bit wide unsigned integer can range from $(00000000)_2$ to $(11111111)_2$ which equals from 0 to 255. Signed integers use most significant bit of storage as sign indicator, cleared bit (0) means value is positive and set bit (1) means value is negative. Negative values are stored by first subtracting one and then in 2's complement form. Below can be found sample values and their machine representation in 8 bits wide signed integer on IA32 platform.

Table 6.1: Signed and unsigned integers in binary form

Decimal Value	Conversion Operation	Machine Representation
128 & over	N/A	N/A
127	127 to binary form (0111 1111), add sign (0111 1111)	(0)111 1111
10	10 to binary form (0000 1010), add sign (0000 1010)	(0)000 1010
1	1 to binary form (0000 0001), add sign (0000 0001)	(0)000 0001
0	0 to binary form (0000 0000), add sign (0000 0000)	(0)000 0000
-1	1 to binary form (000 0001), subtract 1 (000 0000), flip bits (111 1111), add sign (1 111 1111)	(1)111 1111
-10	10 to binary form (000 1010), subtract 1 (000 1001), flip bits (111 0110), add sign (1111 0110)	(1)111 0110
-128	128 to binary form (1000 0000) subtract 1 (111 1111), flip bits (000 0000), add sign (1000 0000)	(1)000 0000
-129 & below	N/A	N/A

Please consider following code snippet:

```
117     unsigned short int iVar;
118     iVar = 0;
119
120     while (1) {                 //Infinite loop
121           ++ iVar;      }       //Increment i continuosly..
```

Figure 6.2: Sample integer-overflowing code

Variable “iVar” is defined as “unsigned short int” which is 16 bits wide in IA32 platform. After initializing to zero, program continuously increments its value. “iVar” is represented with 16 binary digits, its maximum value can be (1111 1111 1111 1111)$_2$, which is 65535 decimal. Next incrementation will result in 65536, or (1 0000 0000 0000 0000) which is 17 bits long and cannot be accommodated on 16 bits. What happens is that most significant bits those cannot fit into 16 bits space get chopped off and remaining result is stored in “iVar”. For this case, (1 0000 0000 0000 0000) will be trimmed to (0000 0000 0000 0000), or (0) decimal. Program will continue looping from 0 to 65535, then wrap to 0 and do the same again.

As a summary,

N-bit wide unsigned integer can have values from (0) to (2^n-1)

N-bit wide signed integer can have values from (-2^{n-1}) to (2^{n-1}-1)

6.1.4.3. Summary of Previous Work

[16] does a great job proposing a class that handles integer operations. Author strongly suggests use of that class for any general-purpose integer handling that is developed in C++. However, there are substantial amount of C code, too, and this paper falls short to address this issue.

[19] and [30] propose a preprocessor application that injects overflow and underflow detection code to the source code. This is fine for retrofitting old code base; however, with a few drawbacks of its own. First, injecting code to the finished product and just passing to the compiler requires great confidence in the correctness of preprocessor application. Second, they increase execution time because of added code. Proposition of [16] also increases execution time, but it is more controlled. After all, developer can decide where to use the SafeInt class, with contrast to automatic tools, which just inject code to check all integer operations.

Compilers usually have command line options to detect assignments to smaller data types. This feature can be used both in debug and release builds. They have also added disadvantage of added code; however, compiler added code is generally much faster and its impact is negligible. As an example, this functionality is provided in MS Visual Studio with “Runtime Checks” option.

There are static code analyzing tools which analyze ready to compile data and generate a report. Those tools are great to increase confidence in the product; however, they cannot be used to guarantee defect free code. They do their best to detect most common scenarios. On the other hand, nested cases can be impossible to detect with those tools. PreFast [41] is a publicly available tool from Research Department of Microsoft Corporation. Advantage of this tool is that it detects other types of code defects as well. However, it is hard to say that this tool is very good at integer overflow detection. Another tool is lint; unfortunately, this tool is showing its age and therefore not competent on finding integer overflow bugs.

6.1.4.4. In new [] Operator

C++ new operator has integer overflow possibility, which occurs during calculation of allocation size. The fact that new operator is universal makes this vulnerability even more concerning. Below is a sample call to operator new and related disassembly of binary code. This code is generated with Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86. Intel(R) C++ Compiler 8.1 generates similar code.

```
122     int *ptr = new int [rand()];

123             call  @ILT+825(_rand) (41133Eh)

124             shl   eax,2  ;This is where integer overflow can
happen.

125             push  eax

126             call  operator new (41146Fh)
```

Figure 6.3: Integer overflow in C++ new operator

The danger here is the false sense of who assumes responsibility. Developers take responsibility when they are using "malloc()" and calculating allocation size manually. However, it is reasonable to expect that compiler will do the math in a safe way when using language features, which is not the case always.

6.1.4.5. Underflows

Integers can underflow, too. This is especially common in loops. For example, if value of "1" is subtracted from an unsigned integer with value of 0, it will wrap to its

maximum value. Underflows mainly caused during string operations (subtracting terminating NUL character from a string length that is already zero) and backwards loop processing. Following code is always defective:

```
127     unsigned int cur;

128

129     for (cur = SOME_MAX_VALUE; cur >= 0; --cur) {

130             ... }
```

Figure 6.4: Integer under flowing sample code

Value of loop variable will always be larger than zero, it is an unsigned type. Variable will wrap to 0xFFFFFFFF, which is a large number, probably larger than the array it is indexing. If developer is lucky enough, there will be an access violation and the program will die. Otherwise, it will execute silently without getting notices and possibly cause a buffer overrun.

6.1.4.6. Conclusion

Integer overflows can happen silently and cause severe security vulnerabilities. Developers should pay special attention while writing code to do mixed mode (signed – unsigned) arithmetic or dealing with rather large values with respect to possible maximum number that the specific integer variable can hold. Best approach, however, would be using template based integer class that makes overflow detection autonomously. Higher-level languages like Visual Basic or C# is therefore very beneficial.

6.1.5. Decision Statements

6.1.5.1. Parentheses (Sev: High, App: Broad)

Always use parentheses in decision statements. Programming languages are very complex in their nature and not every programmer may know every little detail. Using language in the way it makes sense will not work always, there are some issues which their existence in language specification is only because of backwards compatibility and coherence with some (then -like 20 years ago) existing software. C

language is in particular vulnerable to this problem because of being an old language; its root goes before establishment of strong theory of computer languages.

6.1.5.2. A Warning for Comparison Operators (Sev: Med, App: Broad)

A programmer should search for all assignment operators ("=") before each milestone. Below is seen two very simple yet hard to discover errors:

```
131     if (iRequested = iReceived) ..

132     if (iRequested =! iReceived) ..
```

Figure 6.5: Sample comparison operator typos

Line 131 displays an example of typo of comparison operator. Line 132 displays an example of typo in negative comparison (inequality) operator. Simple typing errors like this can jeopardize security of whole project especially in not-so-often-executed error handling codes, which makes detection hard. When a corner case happens and this error handling code is expected to run, it will not (or it will, whichever is worse according to Moore's laws).

One solution to this could be not putting possible left-values (l-value) to the left of comparison. Please consider following code fragment:

```
133     if (iErrorCount = 0)

134             ...

135     if (0 = iErrorCount)

136             ...
```

Figure 6.6: Swapping places of compared variables

Where line 133 will (incorrectly) evaluate to false all the time, line 135 will give a compile time error. As it is repeated frequently in this thesis, a quality conscious programmer always should favor compile time errors and warnings over runtime failures. This is being on the safe side, which is inherent security.

6.1.6. Memory Barriers (Sev: High, App: Low)

Compilers are allowed to optimize code with best of their knowledge of the source code. For instance, please consider following fragment of code:

```
137     Int I;
138
139     I = 4;
140     I = 5;
141     I = 6;
```

Figure 6.7: Memory Barrier Example Part 1

A straightforward code generation could be as follows (in pseudo assembly)

```
142     Read I to Register1
143     Modify Register1 with 4
144     Store Register1 to I
145     Read I to Register1
146     Modify Register1 with 5
147     Store Register1 to I
148     Read I to Register1
149     Modify Register1 with 6
150     Store Register1 to I
```

Figure 6.8: Memory Barrier Example Part 2

Although simple, this code is not as efficient it could be. Please consider following fragment:

```
151     Read I to Register1
152     Modify Register1 with 4
153     Modify Register1 with 5
154     Modify Register1 with 6
155     Store Register1 to I
```

Figure 6.9: Memory Barrier Example Part 3

Compiler optimizes code by removing redundant reads and stores. It can further improve code as follows:

```
156     Read I to Register1
157     Modify Register1 with 6
158     Store Register1 to I
```

Figure 6.10: Memory Barrier Example Part 4

Compilers may optimize the code in a way that skips operation in the source code. This is usually not a problem, since it is not visible to the caller. However, there can be cases, where this is problematic. For instance, program could really want to read, modify and store to this memory address because it is performing a memory mapped IO. To ensure correct operation, programmer must use memory barriers. A memory barrier is a statement, which tells the compiler that contents of memory cannot be cached (It can be changed OOB –by other threads. Alternatively, it must be stored back with most current data because other threads that compiler is not aware of them may depend on it). C and C++ have "volatile" keyword for this purpose.

Alternatively, `#pragma (optimize)` keyword can be used to disable all optimizations locally.

6.1.7. Not Zeroing Unused Out Parameters (Sev: Low App: High)

C and C++ have inconsistent ways of error reporting. Setting last error, returning zero, returning negative values, returning positive values and throwing exceptions are only some of the varieties. Users may get confused with all of these possibilities and forget to check success status. Even worse, programmer can think that he or she is checking return value, although he or she may be checking incorrectly.

Leaving out parameters untouched in an error case can be dangerous if a caller fails checking the success status. Parameters will have some random garbage data causing the program to fail somewhere in the execution process. Microsoft API's do the zeroing right in the beginning of code; this has advantage of checking write access to out parameters in the beginning of code. Although costly, it can help developing code that is more robust.

6.1.8. Call Conventions (Sev: High App: Low)

Order of parameters while passing to functions and responsibility assignment of stack pushes / pops are called calling conventions. Most popular ones are "stdcall",

"cdecl", and "fastcall". Writing code that depends on certain calling convention decreases portability of code and can cause security vulnerabilities.

Please consider following code fragment:

```
159     int sub(int i, int j) {
160         return i-j; }
161
162     int number = 3;
163     sub(++number, number)
```

Figure 6.11: Example Case Where Call Conventions Make Difference

Result of function call is undetermined, because it can be either 4-4=0 (parameters are passed from left to right) or 4-3=1 (parameters are passed from right to left).

Programmer can explicitly define calling convention by declaring it right after function name as follows:

```
164     int __cdecl sub(int I, int J) {
165         return I - J; }
```

Figure 6.12: Example Code of Clarification of Calling Convention

However, this style of coding is not recommended either because of its complexity. Callers should not assume any calling convention and should not modify input variables multiple times while passing to functions.

6.1.9. Improper Size Declarations (Sev: High App: Low)

Please consider following code fragment:

```
166     ZeroMemory(pDecoded, sizeof(HEADER));
```

Figure 6.13: Example for Bad Size Declaration

Nobody can verify the correctness of this piece of code, because it is not possible to tell that "pDecoded" is really pointing to an instance of "HEADER". If the programmer changes the type of variable that is being set, then he also has to traverse

whole code base and change size declarations. This is error prone and cumbersome. Better method is:

```
167     ZeroMemory(pDecoded, sizeof(*pDecoded));
```

Figure 6.14: Example for Better Size Declaration

This approach is good only for pointers of base types. Inherited pointers size would be smaller than what is actually required. Because of its obscure nature, this kind of code defect can be hard to discover, too.

This approach will not work with array pointer in function pointers. Please consider following code fragment:

```
168     void MyFunc(char szString[32]) {

169         ZeroMemory(szString, sizeof(szString)); }
```

Figure 6.15: Caveat in Function Declarations

Here, "sizeof(szString)" will be evaluated whatever the size of pointer in that system is, not to the expected 32. This is because "szString" is just a pointer. Array declaration makes it very confusing, but this type of declaration is needed for type safety. "szString" is a pointer to a place in memory that can hold 32 character variables. In practice, compiler does not care if "szString" is defined to be 32 elements, 4 elements, or empty brackets.

6.1.10. String Constants

6.1.10.1. Automatic String Concatenations (Sev: High, App: Broad)

Strings are concatenated invisibly if two string constants are next to each other.

```
170     printf("Som" "e"

171            " sentence.");
```

Figure 6.16: Automatic string concatenation

```
Some sentence.
```

Figure 6.17: Result of automatic string concatenation

Although this is useful for aligning and making code pretty, its misuse can lead to memory access violations.

```
172     printf("Error message is %s\n   Details are %s and \n%s"
173            "Network Failure",
174            "Corrupt network package is received"   //Comma missing!
175            "This can be due to bad cabling on the network.");
```

Figure 6.18: String concatenation error

In line 174, comma is missing, this causes behind the scenes concatenation of string in line 174 and line 175. "printf" is supplied with two string pointers only, although it is expecting three of them. The result on Windows XP SP2 Build 2600.2158 and with Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86 is

```
Error message is Corrupt network package is receivedThis can be due to
bad cabling on the network.

   Details are n and

ï≡à÷☼ì=É☺Network Failure
```

Figure 6.19: Result of string concatenation error

This surely is not what was expected. The scrambled portion of text is what is on the stack at that time. Imagine for a second that this text message was sent over the network as error reporting mechanism of some sort of server application (web server, application server, etc.). The stack info on the message will be a good advantage for an attacker, which can now infer memory position of this function.

The only proactive solution to this problem can be usage of static code analyzers like "lint", "PreFast" and "prefix" or searching through project files for quotation (") marks and inspecting source code. Either way, this will be long task and may be performed only milestone basis.

6.1.10.2. Escape Characters (Sev: Low, App: Broad)

To begin with, some background information is presented here: Escape character handling is done at C preprocessor level, not compiler level. This means that if

unknown escape sequences are passed to functions, preprocessor will handle those and compiler will be unaware of those and will not generate error.

Preprocessors usually warn about unknown escape characters, a good developer should take those warning account. However, they do not warn against unwanted but legal escape sequences. Please consider following code fragment:

```
176     char a[] = "Result of a/b is";

177     char b[] = "Result of a\b is";
```

Figure 6.20: Unintended escape sequence in strings

Size of array "a" is 17, while size of array "b" is 16. Programmer thought that using '\' is more artistic, however '\b' is escape sequence for backspace character and output for array "b" will be (incorrectly)

```
Result of  is
```

Figure 6.21: Result of sample unintended escapes sequence

Besides erroneous message to the end user, there can be security vulnerability here. Assume that there are some unintended escaped characters. Programmer counts characters with hand (Very bad programming practice) and sends message over the wire with calculated length. This "length" information however will be incorrect because what appears as two characters becomes suddenly one character during preprocessing. Sender procedure, which is instructed to send "x" bytes but supplied only "<x" bytes will send the message, and send whatever is on the stack after that message as well.

6.1.10.3. Avoid String Constants

As a general rule, string constants should not be embedded into the source code, at least they should be avoided as much as possible.

First, software localization staff will have to modify source code itself during localization; modifying checked in source code is not desirable. This will make localization difficult, more time consuming and prone to introducing code defects.

Second, it will bloat the code and will make it harder to read. If there are many long string constants, this will be more significant.

After all, separating resources and code are a very good programming practice.

6.1.11. Octal Numbers (Sev: High, App: Limited)

In C and C++, octal numbers (numbers in base 8) are denoted with 0 (zero) in the beginning. This is not such a good idea, because 0011 is now not 11, but 9. This can be dangerous if programmer uses "0" to align numbers in the source code. [50]

6.1.12. "Struct" Keyword

6.1.12.1. Bit Fields (Sev: Medium, App: Limited- Only Direct Memory Representation Usage)

Bit fields are custom sized fields in structures. For example, it is possible to see a structure with bit fields below, possibly a C representation of a network protocol to communicate with the wire.

```
178     struct MyStruct {
179             unsigned     iMajVersion   : 2;
180             unsigned     iMinVersion   : 2;
181             unsigned     fUnEncrypted  : 1;
182             unsigned     fKeepAlive    : 1;
183             .. };
```

Figure 6.22: Bit fields in C/C++ structures

However, there is a caution for the users of bit field feature: which field is going to be most significant bit, which one is going to be least significant bit is architecture and compiler dependant.

6.1.12.2. Member Alignment

Detailed discussion about this subject can be found in section 8.4.1.2, where we are analyzing interesting compiler flags.

6.1.13. Switch Statements

Switch statements saves the trouble of using bad looking “if” - “else if” - “else” statements. However, it has its own dangers.

6.1.13.1. Auto Fall Through (Sev: Med, App: Limited)

Switch statement has automatic fall through down to other “case” labels unless this behavior is broken with “brake.” This feature is rarely seen useful in real life and has been danger for software quality and security in that perspective. Please consider following example:

```
184     int ReturnExpectedPacketLength(int iPacketType) {
185             int iRes;
186
187             switch (iPacketType) {
188             case PACK_TYPE_AUTHENTICATION:
189                     iRes = 5;
190                     break;
191             case PACK_TYPE_ACCOUNTING:
192                     iRes = 7;           //break is forgotten
193             default:
194                     iRes = -1; }
195
196             return iRes; }
```

Figure 6.23: Forgotten break in switch statement

Here, break directive after Line 192 is forgotten; this erroneously causes the function to return unexpected value. According to P. V. D. Linden [1994]:

> *Default Fall through is Wrong 97% of the Time*
> *We analyzed the Sun C compiler sources to see how often the default fall through was used. The Sun ANSI C compiler front end has 244 switch statements, each of which has an average of seven cases. Fall through occurs in just 3% of all cases.*

This error is an example of “what is not there, although it should have been”. This kind of errors is hard to see if it is not known exactly what to look for. A good habit

before each milestone is string searching each instance of case labels and checking if any of the break statements is missing.

6.1.13.2. Calculations in Case Labels (Sev: Low, App: Limited)

It is not a good habit to construct switch cases with inline calculations like

```
197     case 5 + enum.WeeksAYear:

198             ...
```

Figure 6.24: Calculation in case labels

Although this turns out to be the same binary code as if direct result of calculation is used, it causes poor readability of source code. As mentioned in this thesis, poor readability makes maintenance more difficult, which causes to regressions somewhere along the lifetime of code.

6.1.14. Macro Statements

Although many people frequently discourage usage of macros, macros are so combined with existing code-base. This paragraph aims to help at least correct usage of them, when usage is unavoidable.

6.1.14.1. Spaces in Macros (Sev: Med, App: Limited)

One caveat with Macros is that spaces matter in macro statements unlike regular C code. For instance, please consider following two macros:

```
199     #define MULTIPLY_BY_TWO_A(x)      2*x

200     #define MULTIPLY_BY_TWO_B (x)     2*x
```

Figure 6.25: Macro statement with parameters

They are different.

```
201     int i = MULTIPLY_BY_TWO_A(7);     //int i = 2*7

202     int i = MULTIPLY_BY_TWO_B(7);     //int i = (x) 2*x
```

Figure 6.26: Typo in macro statement

Although this kind of errors usually yields to compile time warnings or errors, there can be some statements, which are legal for C but unintended in runtime.

6.1.14.2. Issues in Macro Calls (Sev: Med, App: Broad)

Macro calls are different from function calls. While calling a function, arguments are evaluated to their final values and then they are passed to the function as parameters. However, during macro calls, arguments are passed in as-is basis. Please consider following program fragment:

```
203     #define MULTIPLY_BY_TWO(ToMul)    2 * ToMul
204
205     int MultiplyByTwo(int iToMul) {
206             return iToMul * 2; }
207
208     int SomeFunc() {
209             int iRes1, iRes2;
210
211             iRes1 = MULTIPLY_BY_TWO(7+2);
212             iRes2 = MultiplyByTwo(7+2);
213
214             printf("iRes1 = %d, iRes2 = %d", iRes1, iRes2); }
```

Figure 6.27: Parenthesis usage in macro statements

Output of this program will be (unexpectedly):

```
iRes1 = 16, iRes2 = 18
```

Figure 6.28: Sample result of bad parenthesis usage

What happened here is operator precedence took over during direct substitution.

```
215     iRes1 = MULTIPLY_BY_TWO(7+2);     //iRes1 = 2 * 7+2;
```

Figure 6.29: Operator precedence during macro substitution

Correct usage of macro should include parentheses as follows:

```
216     #define MULTIPLY_BY_TWO(ToMul)    (2 * (ToMul))
```

Figure 6.30: Correct usage of parenthesis in sample macro

Please note that parentheses are inserted not just around macro parameters, but also around entire macro, too.

6.1.14.3. Macros are Type Unsafe (Sev: Med, App: Limited)

C and C++ compiler checks parameters for type safety when passing to functions. However, passing arguments is just substitution and occurs before compilation during preprocessing. This is a very bad situation, callers use libraries and they seldom refer to source code; what they have usually is only function or macro name. Please consider following (though practically useless) implementations of pointer iterators and how misleading the macro name is:

```
217     #define ADVANCE_TO_NEXT_BYTE(x)   ((x)+1)
218
219     BYTE * AdvanceToNextByte(BYTE * b) {
220           return b+1; }
```

Figure 6.31: Sample type-unsafe macro

The macro version is obviously designed for pointers of type BYTE only. However, caller of this macro will not be warned in case of passing different type of pointer. Assume that caller passes a pointer to an instance of type WORD (two bytes); each iteration will be now two bytes, instead of intended single byte. Function version (Line 219~220) of this implementation would warn and suggest type casting if it is really what was intended. This macro could be written with explicit type cast, but programmer probably did not think this (probably future) usage. This unexpected extension of usage scenarios are one of the most dominant causes of security vulnerabilities.

6.1.14.4. Summary

We consider usage of macros generally unnecessary and they must be avoided if possible (and it should be possible almost all the time). Motivation to use macros includes their speed and flexibility on accepting arguments of different types.

Functions can be inlined, too. This allows lightweight functions that are as fast as macros because they do not have stack operations. Although programmer can define explicitly with "inline" keyword, C / C++ compilers are good at detecting frequently used short functions and in lining them.

The answer to the second argument (flexibility) is usage of template functions in C++. They are type-safe and provide almost same amount flexibility.

The only correct usage of macros may be (though, arguably) in the case of unavoidable need to preprocessor string concatenations ("##" operator), where C/C++ language does not offer an alternative.

6.1.15. Unexpected Compiler Optimizations

What is written as high-level source code does not necessarily translate exactly to binary code; compiler optimizations can change order of operations, delete them or add new operations, unless "observed" behavior of source code does not change. For instance:

```
221     int Function() {
222             int i, j, k;
223
224             i = 2;
225             j = 3;
226             k = 6; // ← Does not have effect on the result!
227
228             return i * j; }
```

Figure 6.32: Sample code with optimized out code lines

Observed behavior of this function is returning 2*3=6. Once compiler deduces this fact, it can

- Optimize out assignment 6 to k
- Optimize out stack space allocated for k
- Optimize out real assignments to “i” and “j” and use registers instead
- Optimize out real multiplication and replace it with constant “6”
- Optimize out function body, and replace entire function with constant “6” according its capabilities and configuration.

Security related clean-up operations are especially prone to this optimizer side effect since some security calls are seemingly unnecessary. For example:

```
229     void DecryptFile() {
230           BYTE baKey[32];
231
232           ...
233           if (SUCCEEDED(GetUserKey(baKey, sizeof(baKey))) {
234                 ...
235                 ZeroMemory(baKey, sizeof(baKey)); }
236           ... }
```

Figure 6.33: Sample optimized out security code

Here, developer gets user key, uses it and when it is required no more, wipes it out. Actual behavior is probably not as described in previous sentence. If optimizer deduces that source code assigns zeros to local array “baKey” and never uses these values until it goes out of scope with function return, it can decide that this is not necessary because it does not affect observed behavior. Developer should have used “SecureZeroMemory” [67] API call instead.

6.1.16. Obscure C Syntax

Stay away from uncommon and obscure syntax of C. Using uncommon practices will be confusing, thus error prone.

6.1.16.1. Array Declaration

In C, the following two lines are equal:

```
237     int iaVar[SIZE];

238     int SIZE[iaVar];
```

Figure 6.34: Obfuscated C array declaration

This is unnecessary syntactic redundancy. Current C compilers from Microsoft and Intel no longer allow such array declarations.

6.1.16.2. Concatenated Operators

Developers should always avoid usage of concatenated operators since they are very confusing and therefore make code error prone. Examples would be

```
239     int i;

240     int k = 10;

241

242     i = -++k;     //(k plus 1) negated

243     i = ++i;

244     i = i+++k;    //Confusing: Which one is incremented? i or k?

245     i = i++-k;    //Same problem, which one is incremented?
```

Figure 6.35: Confusing operator usage

6.1.16.3. Comma Operator

Comma operator in C/C++ is used to merge multiple operations in one syntactic operation. For instance, such a code is valid C++ code:

```
246     func(iParam1, (i=rand(),--i, i <<= 2));
```

Figure 6.36: Sample code using C comma operator

“func” is a C++ function taking 2 parameters. First parameter is what “iParam1” variable happens to hold, second parameter is a calculation performed as assigning “i” a random value, decrementing it and left shifting it twice.

Although it allows interesting operations to be done in limited space, its usage is discouraged since it makes code harder to read and confusing. For instance, please consider following example:

```
247     i = k = 10, k-=2;
```

Figure 6.37: Sample confusing code using C comma operator

It is hard to tell if "i" and "k" are assigned to 10 and k is subtracted by 2; or if "k" is assigned to 10 and subtracted by two and assigned to "i". What actually happens is first case, at the end of these operations, "i" will have value 10 and "k" will have value of 8.

6.2. Function Level

6.2.1. Formatting and Commenting

6.2.1.1. Code is for computers, format and comments are for humans. Since code is still written manually by humans, correct usage of formatting is very important to ensure correct human behavior: That is correct coding. We state that correct and high quality comments will help raising confidence level during the fixes in maintenance phase. We support this statement with a survey and provide guidelines for high quality comments.

6.2.1.2. Approach to the subject

Author prepared a survey for investigation of formatting and commenting usage among developers. This research is done to support below discussion.

27 professional developer has attended the survey. Questions and reply distributions are below.

- Did you ever find a bug thanks to good formatting of code
    - Yes, a lot (19)
    - Yes, only a few times (5)
    - No (3)
- Do you try to write highly readable code during development

- Always (17)
- Sometimes (10 –Attendees commented that they write readable code depending the importance of the code. This habit is very bad indeed, because any single line of defective code reduces quality and can cause vulnerabilities.)
- No (0)

- Do you believe that you write enough comments while writing the code
  - Yes (13)
  - No (14)
- Do you write comments only to satisfy requirements
  - Yes (2)
  - Mostly (4)
  - Sometimes (10)
  - No (11) (Most teams did not have such requirement guidelines)
- Do you think there are enough comments in the code you are working on
  - Yes (18)
  - No (9) (Interestingly, attendees who replied as "no" were working in sustained engineering groups, while most of the "yes" were working in development groups.)
- Have you ever noticed that you have written a code with security vulnerability
  - Yes (8)
  - No (19) (We don't believe that most of the attendees had qualified knowledge of distinguishing regular bugs from security vulnerabilities.)
- Did anybody else noticed that you have written code with a security vulnerability
  - Yes (5)

- No (22)

- Do you think that correct formatting and commenting can prevent security vulnerabilities
  - Yes (20)
  - Sometimes (7)
  - Negligible (0)

Results of this survey are discussed in the next sections.

6.2.1.3. Formatting

Survey revealed that developers try to write code with correct format and they believe that correct formatting reveals code defects more easily.

Formatting is a taste that differs from developer to developer. Although there are standards for commenting, habits mostly shape the output of code. It is a good thing that development environments help enforcing a standard in formatting by automatically updating written code.

A very important aspect of formatting is consistency. Tricky indents or parentheses can easily turn into illusion because of accustomed eyes.

```
248     if (...) {
249             ab...
250             cd... }
251
252     some_op..
253
254     if (...) {
255             ef...
256             gh... }
257
258     some_op..
259
260     if (...)
261             ij...
262             kl...
263
264     some_op..
```

Figure 6.38: Example of Vulnerability Caused By Bad Formatting

In the previous code snippet, line 262 is not in the “if” block, though it seems so. This can cause a security vulnerability, if, for example, “kl” is used to hold buffer size.

Formatting should be done in a way that prevents increased visual size of code. If the code gets longer and longer, important parts of code fall apart; most notably variable declarations and usage of them.

A less known habit is using spaces instead of tab characters to indent code lines. Advantage of this is preventing format brake in different tab sized environments.

6.2.1.4. Commenting

Unfortunately, same comments as formatting are not valid for commenting. Developers want to see more comments in the code they are working on, but they do

not take care of it themselves. Some of the attendees replied that they are writing comments because they have to. From this response, we can conclude that they do not care about comment quality enough.

Basic principle of commenting is writing just enough and informational correct comment. Following example gives an example of bad commenting.

```
265     //declare integers
266     int i, j, k;
267
268     //assign values
269     i = 2;
270     j = 3;
271
272     //add two integers
273     k = i + j;
```

Figure 6.39: Example for Bad Source Code Comments

Only repeating obvious operations is not good at all. A better commenting practice is shown below:

```
274     //Initialization of this member is compulsory, see MSDN
275     OsVersion.Size = sizeof(OsVersion);
276
277     //OsVersion is cast to ex version,
278     //This is safe as documented in the manual at
"GetVersionInfoEx"
279     GetVersionInfoEx((OSVERSIONINFOEX *) & OsVersion);
```

Figure 6.40: Example for Better Source Code Comments

A good comment should be written keeping in mind that reader of the comments will be most probably somebody else. Moreover, that will most likely happen during maintanance phase and therefore reader will be quite strange to the code Because of that, good comments must mention about

- dependencies (we think that these are most valuable comments that are at the code level),
- reasons of unclear decisions,
- summary of historic bugs,
- high risk zones,
- possible race conditions (we think that these are most valuable comments that are at the function level),
- performance considerations.

6.2.1.5. Additional Discussion about the Survey Results

We also noticed that requirements about comment count only bloats the code without adding any useful information. Good commenting can hardly be enforced. We believe that commenting must be thought. One of the best ways (and possibly time consuming) is sending developers to sustained engineering groups for hands on training.

Although not shown neither in questions nor replies, most of the attendees admit that they write comments concurrently with the writing of code and never visit them back. Although concurrent commenting has significant advantages, they quickly diminish if they have never visited back.

6.2.2. Kernel Mode Access Checks

An unhandled exception in kernel mode will cause system crash, i.e. blue screen in Windows, core dump in Linux. No user process should be allowed to manage to bring kernel to undefined state and cause crash. However, if user mode input values are not validated and access to them are not guarded, this can happen.

Please inspect following example:

```
280     BOOL GetSystemPageSize(unsigned int * puiRetVal) {
281           if (NULL == puiRetVal)
282                 return false;
283
284           try {
285                 if (GetKernelPageSize(puiRetVal) == FALSE)
286                       return FALSE; }
287           catch (...) {
288                 return FALSE; }
289
290           //Validity of puiRetVal can change after executing
previous
291           //code but before executing next code.
292
293           //Assuming system page size is twice as kernel page size
294           *puiRetVal *= 2; }
```

Figure 6.41: Sample vulnerable kernel mode code

If caller is just a bad programmer passing corrupt pointers, then this code will work just fine. On the other hand, if the caller is planning an attack to kernel and trying to crash the system, he will certainly succeed. Please note that developer checked for validity at the beginning of the function by guarding with exception handling mechanism. However, on line 293, out parameter is modified without exception handling guard. If the attacker manages a race condition and changes status of that page during check and reuse, it will manage to crash the system. He does not need any privilege; just executing a program will cause system crash.

6.2.3. Exception Safety in C++ and in C with SEH

If used properly, exceptions are good at reporting errors noisily and timely. However, broken execution path can have negative impact on completeness of transactions.

6.2.3.1. Main Problem

It is very difficult to know where an exception can be thrown. There are C++ annotations to declare functions as throwing or not throwing exceptions. It even allows specifying what type to throw. However, even modern compilers do not give importance to this feature and do not enforce compliance even if exception specification are present in the code. Static code analyzers could use this information; however, they are not good at that either if throwing level is deeply nested or in other modules, of which source code is not available. Worse, C language does not even have that feature. Maybe worst case, operations like division or floating-point arithmetic can raise exceptions even there is no function call visible.

Unexpected exception throws result in broken flow of code execution, which leaves class states in unexpected states. It might be possible to design classes stateless (i.e. idempotent), but designing function internals that way is very difficult and sometimes not even possible.

6.2.3.2. Case Study

For instance, please see following code snippet (Assume that operator new is set to exception throwing mode):

```
295     class String {
296     private:
297             char * szStr;
298     public:
299             ...
300             String& operator = (char *szRight) {
301                     size_t cRightLen;
302                     cRightLen = strlen(szRight);
303                     delete [] szStr;
304                     szStr = new char[cRightLen + 1];
305                     strcpy(szStr, szRight); }
306     };
```

Figure 6.42: Example for Exception Safety

Line 303 deletes old string pointer and line 304 creates a new one. If there is not enough memory at that time, “new” will throw an exception. After that point, object of string class will be in undetermined state.

6.2.3.3. Prevention

There are several methods to prevent this from happening. Most straightforward method is having an exception handler in the function. Handler will ensure the completeness of transaction and then re-throw the exception to its caller. This method will increase code size unnecessarily by putting exception handler code. Furthermore, to allow this method work, most of the used variables need to be assigned initial values, which increases code size and runtime cycle consumption even more. Unfortunately, C++ language does not have a “finally” keyword. That keyword, which is present in some languages and in Windows OS Structured Exception Handling mechanism, allows adding checkpoints to the code. Checkpoints are always executed upon exit of guarded code, it is guaranteed by the language or operating system. “Finally” blocks are mostly used to guarantee proper clean up.

A better method is rearranging function calls to make functions inherently exception safe. This method is best solution if it is possible and done properly. However, developer must be very careful to ensure correct order.

Another approach is taken by C++ template library. In essence, that method consists of creating a temporary object, successfully constructing it, and then replacing its contents with a non-throwing member function. Following code is a demonstration of this method:

```
307     class String {
308     private:
309             char *_szStr;
310
311     public:
312             String(const char *szStr) {
313                     size_t cLen = strlen(szStr);
314                     _szStr = new char[cLen + 1];
315                     strcpy(_szStr, szStr); }
316
317             ~String() throw() {
318                     delete [] szStr; }
319
320             void Swap(String &strOther) throw() {
321                     std::swap(this->_szStr, strOther._szStr); }
322
323             String& operator = (const char *szStr) {
324                     Swap(String(szStr)); }
325     };
```

Figure 6.43: Example for Exception Safety Improvement

Line 324 can throw an exception, but it is not important. If creation of temporary object finishes without throwing any exception, non-throwing "String::Swap" operation is called and contents of that string is acquired. Old contents of old string are pushed into temporary object, which guarantees release of old resources. Please note that destructor is no throwing as well. Although very elegant, this method is not applicable to every case, notably for the functions, which are not involved in object creation.

6.2.4. Function Reuse

6.2.4.1. Reusing Code whenever Possible

Verifying correctness of a function is difficult; verifying correctness of two functions is even more difficult. Reuse code whenever possible; try not doing same thing twice. If there is a sequence of operations repeated in different locations of source code, put those sequence of operations in a function and call that function. This will make change applying easier and prevent omitting changes.

On the regressions side, this approach can be good or bad. Now there will be more dependencies to functions overall the source code. If internal logic of one function is changed, this will effect whole application, which can be bad or good depending on the circumstances. This change will be reflected uniformly to whole application and all dependents will be updated accordingly. However, code not expecting such a change can be effected in a bad way and this cause regressions. To prevent this from happening, each function should be designed according following principles [40]:

◊ Do one thing and do it well:

This idea is motto of UNIX community. Functions should not overwhelm themselves by trying to do more things it is supposed to do. Otherwise, it will complicate error handling and rollback in case of exceptions. Serviceability and maintenance will be easier. Moreover, reusability of the code will be higher.

◊ Functions should be black box:

What is meant here is that functions must be transparent to the input and users should not be required to know internals of functions. Only API reference should be enough to fully use the function. For instance, if function allocates memory, users should not be kept responsible of keeping track of those resources. We understand that this can be difficult with C++; however, automatic pointers help with resource collection.

◊ Decorate non-idempotent functions:

Idempotent functions do not change state of program. Their next output is independent of previous calls (How many times and with which parameters it is called). On the other hand, non-idempotent functions bring the program to a new state. They must be processed as transactions, complete commit or rollback

mechanisms should be implemented. To make users aware of nature of functions, a naming convention can be used. Best naming convention is probably using “const” identifiers in C++.

6.2.4.2. Use Default Parameter Values instead of Function Overloading

C++ and some other languages allow defining some of the parameters of a function default at the compile time. If compiler sees that the programmer is not supplying a parameter, it uses default values.

For instance, on the sample below, “OpenHttpServerPort” function can take configuration options on different granularity. To support this, four different versions (Lines 326, 328, 331 and 334) of the almost same code are rewritten. Functions with less parameters (coarse granular) obviously have predefined default variables to be used when opening socket (because, operating system will request some value). These functions can easily be merged into one single function using default parameter values.

```
326     bool OpenHttpServerPort(
327             const int iMaximumCalls);
328     bool OpenHttpServerPort(
329             const int iMaximumCalls,
330             const unsigned short usLocalAddressIndex);
331     bool OpenHttpServerPort(
332             const int iMaximumCalls,
333             const unsigned short usLocalPort);
334     bool OpenHttpServerPort(
335             const int iMaximumCalls,
336             const unsigned short usLocalAddressIndex,
337             const unsigned short usLocalPort);
338
339     bool bool OpenHttpServerPort(
340             int iMaximumCalls,
341             const unsigned short usLocalAddressIndex = 0,
342             const unsigned short usLocalPort = 80);
```

Figure 6.44: Reducing function matrix with default parameter usage

A parameter that is considered to be assigned a default value must really have a reasonable default value that will not change. For instance, giving interest parameter a default value in a tax return calculation function is probably a bad idea, because interest values change yearly. A programmer might think that this function has default parameter and might not supply current value resulting in wrong calculations, which are very hard to figure out. What is not seen is hard to debug. Moreover, it is very important to set default values to be compatible with all possible other parameters. Since callers are allowed to leave them blank, they must have meaningful values.

6.3. Software to Write Software

Tools used for software development present tremendous amount of importance for high quality releases. On the extreme (and unlikely nowadays) case, low quality compiler will produce erroneous code. Tools can make certain tasks much easier (or even possible) compared to doing by hand.

6.3.1. Development Platform

6.3.1.1. Integrated Development Environments

On the very old days, programmers were punching holes in cards to write codes. Fortunately, these days are long over and there are many sophisticated tools for programmers. These tools are combining many facilities a developer would need during implementation phase of a project. Therefore, they are called integrated development environments, or shortly IDE. A good IDE should provide:

A good source text editor with

- Code highlighting to make it easier to distinguish different elements of code. Normally, in common environments, only keyword highlighting is present. However, highlighting string constants in the source code is very important, too. On the other hand, a program called SourceInsight parses the code as a compiler does, and highlights, underlines, italicizes, and makes bolder to a very granular level.
- Basic syntax checking; this will cut unnecessary compiling times to learn that there is no keyword in C called “strct”.
- Warnings against deprecated API’s.
- Simple yet useful features like source code commenting / un-commenting, auto indentation, and style checking, etc. Furthermore, it must support code auto formatting as well. Some tools allow to format a code to a predefined format.
- A fast, optimizing (both for space and/or speed) compiler capable of producing correct and meaningful error/warning messages and capable of producing absolutely error free binary file representation of supplied source code. Run time code checks are a very welcome improvement. We do not

expect too much from an IDE's built-in compiler. After all, release bits can be compiled with a compiler of custom selection.

- A fast, optimizing linker.
- A powerful debugging engine capable of source code and assembly debugging with use of full symbols and modifying source code during debugging.
- Integrated and complete help for IDE itself, programming language and supplied libraries.

A good IDE with above-mentioned features will keep a programmer concentrated to its job by making tasks shorter, more intuitive and easier.

6.3.1.2. Simulators

The earlier the code is tested, the better it is in terms of quality and costs economy. Some software projects will require specialized hardware to test it. However, this hardware could be very expensive to dedicate one to each of the programmers / testers; or it may not be available until later phases of the project. Using a simulator to increase the number of testers or to begin testing earlier is advised. However, simulators present many weaknesses.

First, they are slower than the original hardware. This will give wrong estimates of performance and may hide some of the race conditions.

The correctness and exactness of simulator are essential, yet it is hard to test and verify; testing the code against incorrect simulator would cause very unpleasant surprises to the end of the project. Testers using distrusted simulator will end up running tests against both the simulator and actual hardware, just time consuming and result confusing.

6.3.1.3. Profilers

Readers might wonder why profilers are mentioned in a thesis about secure programming. Over optimizing or optimizing in wrong places manipulates the code extensively and can result in code defects. As a rule, each code defect can end up being security vulnerability.

A project should be optimized during requirements phases by cutting off unused features, unneeded flexibility or unneeded scalability. This also reduces attack surface in the future. Design phase also presents opportunities like choosing better algorithms or defining synchronization bottlenecks better. During implementation, optimization can be done at two levels: algorithm and source code.

May be the best summarization of choosing more complex algorithms is stated by Rob Pike in his Notes on C Programming:

> *Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.*
>
> *Rule 2. Measure. Don't tune for speed until you have measured, and even then don't unless one part of the code overwhelms the rest.*
>
> *Rule 3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)*
>
> *Rule 4. Fancy algorithms are buggier than simple ones, and they are much harder to implement. Use simple algorithms as well as simple data structures.*
>
> *Rule 5. Data dominates. If you've chosen the right data structures and organized thongs well, the algorithms will almost always be self evident. Data structures, not algorithms, are central to programming.*

Optimizing a code at the source code level, on the other hand, is usually harder and less efficient. Over-optimizing a piece of code is in particular very dangerous. Hand optimized code gets harder to understand, prone to bugs and very rigid. They usually have many predefined limits and constants, several assumptions and shortcuts; these are generally traits of fragile source codes. It is very reasonable to fine tune performance bottlenecks even at the assembly level to squeeze every possible CPU cycle; however, this should be done only if it is proven to be needed. A working implementation should exist in order to be able to prove, otherwise, there would be only guessing; a method that has no place in a serious engineering practice.

Profilers can help developers to have a profile of their application. With better picture of bottlenecks and regions of code that needs optimization, a developer can pinpoint what to optimize and how much to optimize. Most likely, after setting profiler, developer will perform scenario based testing; this could be handing off the application to a real life end user or mounting it to the environment that reflects the environment the application will be running on once shipped. After several hours of data collection, profiler will give results. After correctly analyzing the results, developer will less likely tend to over optimization.

6.3.1.4. Sand Boxes

Sand box is an isolated private space for a piece of code. A sand box will provide an environment, complete set of global dependencies and local dependencies. That piece of code will “feel” like it is in real life executing in the middle of the process. This capability is very useful to test functions in customized scenarios. Sometimes, it can take long time to bring an application to desired state. For instance, assume that memory stress handling code is subject to test. Without using any tools, a real stress scenario can be realized, which normally requires ample resources. On the other hand, isolating that piece of code and giving it an environment suffering from memory shortage will make testing much easier.

6.3.2. Debuggers

6.3.2.1. NTSD / CDB

NTSD and CDB are two Microsoft provided very similar command line debuggers that use same debugger engine. They are extremely powerful and provide everything a debugger can provide. They are updated frequently and are available for different platforms. They come with a very good, actually helping help file, which demystifies many hard-to-understand features of these powerful engines. On the downside, however, command line user interface is not attractive to many users and makes usage harder than GUI tools.

6.3.2.2. WinDBG

WinDBG is using same debugger engine as NTSD and CDB, however it provides a graphical user interface. Although user interface is not very exciting nor intuitive, it still provides the flexibility of NTSD/CDB in an easier to use environment.

6.3.2.3. Symbol Files

Symbol files include type, address and line information of source files. When used with a debugger, it can help debugger to provide resolved stack information (call stack, local parameters, return values) and line number in the source of current code.

Not all symbol files present same amount of information. While private symbol files will generally provide detailed information about binary, release editions of symbol files will include only parameter names, not the types.

Full symbol file generation is crucial for in the field debugging. Otherwise, debugging will require huge amount of disassembly and heuristics.

Microsoft has a public symbol server on the web. To further analyze system calls and call stacks, environment variable for symbol server (_NT_SYMBOL_PATH) can be provided as:

*srv*c:\cache*http://msdl.microsoft.com/download/symbols;*

6.3.2.4. Effective Usage of Debuggers

Although effective usage of debuggers is an important skill in secure software development, it is beyond scope of this thesis. However, readers are strongly encouraged to develop their debugging skills if they do not feel competent and comfortable. Debugging will open a new door to the internals of binary, which after all executes in the machine.

6.4. Libraries

6.4.1. Motivation

Developers use libraries to increase code reuse and cut from development time. Taking advantage of existing functionality is good idea unless that functionality does not bring its security threats with it. There is a saying that goes, as "Being able to ask is half of knowing." If developer is not aware of the potential vulnerabilities in the libraries that are used, otherwise secure code could be poisoned with external code.

Aim of this section is not being a substitution for the documentation of those specific libraries. Such a goal would be repeating old work and would not provide any useful data. Rather, the goal of this work is stressing out deficiencies of some highly popular C/C++ libraries. Sometimes, usage of a certain library is unavoidable; this work also gives information how to use possibly insecure libraries safely.

6.4.2. Previous Work

Documentations of specific libraries are good resource for the capabilities of that library. However, some of the popular libraries are old and therefore their documentation lacks information about secure usage of those libraries.

Design Patterns [14] gives information about how to adapt old behavior to the desired new one. However, that book was obviously written without security in mind, and it can be difficult for the reader to pinpoint specific tips. Design Patterns section of this thesis covers that shortage and it can be a good reference while reading this section.

We are unaware of any related work about this subject in academic environments.

6.4.3. Correct Thread Model

6.4.3.1. Single Thread

Single threaded libraries are not designed for concurrent calls. They are not reentrant and depend on shared global variables. Concurrent calls will corrupt internal state of library and will result in errors. However, they are usually faster than their multi-thread capable counterparts are, because they are not overwhelmed with synchronization code. Usage of single-thread capable libraries is discouraged; there may be functions that create different threads under the hood. For example, calls to COM, ODBC and RPC will create threads not visible to programmer, and their callback will stress single-threaded libraries since they will be happening on some other concurrent thread.

6.4.3.2. Multi Thread

Multi-threading capable libraries are a little bit slower than their single-thread capable counterparts are. However, they are the good way to go if the programmer cannot be sure that the program is and will be only single-threaded.

6.4.4. Private Libraries

6.4.4.1. Input Trust Decision

All public input to libraries should be considered unsafe. Private calls with internal data, or validated public data can be considered in the trust domain, thus input can be taken safe. However, problem is forgetting boundaries of trust, i.e. which function is private, which one is not.

One suggested method to even distinguishing private and public entry points is decorating private functions, such as prefixing with underscore “_” or “p_”. This will

help ringing bells when passing not validated input to some private function. Public functions (without this decoration) should validate each and every one of input parameters before passing to (decorated) private functions.

Public functions passing not validated input to public functions are delegating responsibility of validation, which can be considered safe if called public function is written safe. It is reasonable to delegate such a responsibility (for performance reasons, validation can be time consuming and duplicate validations can cause unnecessary performance drops) provided that ownership of these responsibilities is well known.

6.4.4.2. Memory Checking

Windows API has several calls for checking validity of supplied pointers. They are listed below:

```
343     BOOL IsBadReadPtr(
344       const VOID* lp,          //Pointer to start of region to
check
345       UINT_PTR ucb);           //Length of region to check
346     BOOL IsBadWritePtr(
347       LPVOID lp,               //Pointer to start of region to
check
348       UINT_PTR ucb);           //Length of region to check
349     BOOL IsBadCodePtr(
350       FARPROC lpfn);           //Pointer to memory
351     BOOL IsBadStringPtr(
352       LPCTSTR lpsz,            //Pointer to start of region to
check
353       UINT_PTR ucchMax);       //Length of region to check
```

Figure 6.45: Pointer validity checking with Windows API

These API calls can be used to verify pointers from third party libraries or user that is not trusted. However, the problem with these functions is that they are not thread safe (as documented in Microsoft Developer Network). What they are doing is not magic. For instance, it is easy to see what "IsBadReadPtr" doing from debugger

disassembly; it is trying to read 1 byte from each page of memory region in the guard of exception handling. If there is exception, it returns TRUE. Else, it exits from read loop and return FALSE. If there is another thread changing access permissions or freeing parts of memory, this function may not detect it. Correct usage of this function should involve critical section usage.

Memory access check is, again, state handling problem. A function should not depend to any state of any pointer. It should try whatever it is supposed to do, if there is an access violation, virtual memory hardware will raise an exception and it will be caught whatever function is responsible. Such a stateless function should be written as exception safe, i.e. it should work as commit or rollback. It should take account that every line can possibly cause exception and if there is one, it should leave by just rolling back, rather leaving dependent variables in an undefined state. This thesis has mentioned about exception safety before, reader are urged to refer to that section if they have not done it so. Also [38] has good information about exception handling.

6.4.5. C Runtime Library

C is an old language and it shows. Its widely accepted usage was well before establishment strong fundamentals software engineering practices. During standardization, committee had to stick on bad practices for the backward compatibility. Therefore, some of the function interfaces are unsafe in C standard library. For instance, strcpy() may overrun its destination buffer if source pointer points to a string that is longer than destination buffer, because there is no way to know length of destination buffer and stop copying. Similarly, strcat () and gets () may overrun their destination parameters.

C runtime library functions have Unicode and multi-byte counterparts in Microsoft CRT; for instance, “strcpy” has two other variations: “wcscpy” for Unicode string copy and “_mbscpy” for multi-byte strings copy. These variations are unsafe, too. All variations unsafe functions in CRT are banned in major software companies and all new projects should declare them as deprecated in a header file; this will warn careless programmers against their accidental usage.

6.4.6. String Safe

Windows SDK has a new set of string manipulation library, “String Safe”. This library declares many functions with enhanced buffer security and intended to replace the functions coming with C standard library [68]. A sample function will be examined to show approach and capabilities of this library.

```
354     //C Style String Copy
355     char* strcpy(
356        char *strDestination,   //Destination Buffer
357        const char *strSource); //Source Buffer
358
359     HRESULT StringCchCopy(
360         LPTSTR pszDest,        //Destination Buffer
361         size_t cchDest,        //Length of Destination Buffer
362         LPCTSTR pszSrc);       //Source Buffer
363
364     HRESULT StringCchCopyEx(
365         LPTSTR pszDest,        //Destination Buffer
366         size_t cchDest,        //Length of Destination Buffer
367         LPCTSTR pszSrc,        //Source Buffer
368         LPTSTR *ppszDestEnd,   //Ptr to End Of Destination Buffer
369         size_t *pcchRemaining, //Ptr to Buffer for Remaining Space
370         DWORD dwFlags);        //Option Flags such as
371                                //     STRSAFE_FILL_BEHIND_NULL
372                                //     STRSAFE_IGNORE_NULLS
373                                //     STRSAFE_FILL_ON_FAILURE
374                                //     STRSAFE_NULL_ON_FAILURE
375                                //     STRSAFE_NO_TRUNCATION
```

Figure 6.46: String-safe API example

As seen above, regular strcpy () is replaced with StringCchCopy () which takes an additional parameter for destination buffer length to prevent possible buffer overruns.

However, there is also a more power version of this function that also enables further actions on error conditions.

String safe library functions have two variations, one variation has “StringCch” prefix while the other one has “StringCb” prefix. “StringCch” functions measure length of string as item count, namely the count of available Unicode or ASCII characters. “StringCb” functions, on the other hand, measure the length with available byte count, which is different for Unicode strings; a Unicode buffer with 100 bytes of available space can hold up to 50 Unicode characters. A good programming practice is using only “StringCch” functions and disabling “StringCb” functions with a “#define” statement before including String Safe “strsafe.h” header.

Complete reference can be found on [42].

6.4.7. C++ Standard Template Library (STL)

STL is a safe, fast, good designed, easy to use library for C++ users, and it is standard. Only caveat with it is that some implementations are very obscure and really hard to debug. Some implementation (until recently, Microsoft’s for example) use internal naming schemes “_v” for all values and “_p” for all pointers which makes it very difficult to understand what is “_v” and what does point “_p”. Though implementation might be considered bug-free, this trait makes it harder to debug own code. However, this is only implementation dependent and not a fault of standard.

6.4.8. Active Template Library (ATL)

ATL is a Microsoft provided library consisting of helper functions for its COM technology and template functions for several data structure implementations.

COM helpers and wrappers are really useful when developing COM applications as they reduce code load of the developer and implement repeated bulk parts of COM codes. Especially, threading models of COM is made very easy to program thanks to ATL. Moreover, Microsoft IDE Visual Studio has many wizards to help using ATL.

There is nothing very interesting about template data structures. However, simplicity and clarity of their implementation must be credited if compared with template data structure functions of STL. They are not designed to be perfectly object oriented, therefore some less frequently code is not present.

6.4.9. Microsoft Foundation Classes (MFC)

It is mainly a wrapper library among Microsoft Windows 32 user mode API's and comes with Microsoft Visual Studio IDE. It gathers related API calls in object oriented fashion and therefore hides some of the inconsistencies of Win32 (They are there mainly because of backwards compatibility issues going back to eighties) API. Moreover, it encapsulates some rather error prone and difficult tasks and presents a clean easy-to-use interface. It has been preference in the field for a long time.

Although it is thoroughly tested by library developers and in the field, vulnerability was found in its ISAPI parser libraries. However, it is assumed that this should be an isolated case and library is considered safe.

6.5. 64 Bit

Previous work on this subject is mature. This thesis does not aim to repeat that work here, rather its existence as a security threat is presented.

32-bit and 64-bit mixed code should be written very carefully since assignments and pointer calculations can yield to incorrect results. [19] does a great job proposing an automatic tool for detection. Furthermore, [41] is a very useful, publicly available product that performs source code level passive checking of 32/64 bit incompatibilities. [11] can be used to detect any misaligned pointers during runtime; however, this is possible only for debugging builds.

7. Verification

7.1. Preventive Measures

7.1.1. Assertions

Electronic circuit schematics have test points with reference data. They are used to pinpoint defective electronic element. Idea of such test points, or assertions of certain locations of code came first from Alan Turing [46], and they are developed by Hoare and Dijkstra [45].

```
376     class String {
377     private:
378             char * szStr;
379     public:
380             ...
381             Empty() {
382                     assert(szStr != NULL);
383                     delete [] szStr;
384                     szStr = NULL; }
385     };
```

Figure 7.1: An Assertion Sample

Designer decided that performing “Empty” operation on an already empty string is not correct and should be avoided. Therefore, developer added this assertion at line 382 to alarm if there is such a request. If execution gets past of this checkpoint, it is safe to assume that “Empty” operation is being done on a valid string object.

Interesting thing with assertions is that they have volatile nature of being in the actual binary. Mostly, “assert” blocks are designed to be removed in optimized builds, since they are expected to have done their job during testing of the product

with debug version. This behavior is important to note, because an important check encapsulated as assertion will be removed in release build leaving code without a necessary check. [37] has a complete chapter that focuses on correct usage of assertions. Since it is considered one of the best articles about this subject, this thesis will not focus on correct usage of assertions. However, main point is using assertions to successfully develop trustworthy applications.

7.1.2. RockAll Memory Manager

RockAll is a memory manager that is designed to perform all memory requests of an application. It can switch to page-heap mode in runtime. Furthermore, it can detect majority of buffer overflows. This is done by injecting security cookies after memory blocks and checking those cookies on release of that memory piece. Some versions of C runtime library do the same, but this behavior is not standard.

RockAll can be used to monitor and profile memory needs of an application since it can generate reports at the runtime. This memory manager can be used to determine near future needs of the application.

7.2. Testing

7.2.1. Structural Tests

7.2.1.1. Fuzzing

Fuzzing is a method to test entry points of functions or modules. Every entry point expects data according a predefined interface definition. With fuzzing, slightly modified packets are sent to the entry points to check their error handling capabilities and robustness.

Simple fuzzers are just network noise generators; they do not understand underlying communication protocol. Many protocols have some sort of checksum control and simple network noise causes fuzzed packets to be discarded in the very early stages of parsing. Fuzzers, that are more complicated, generate or modify packets in the knowledge of implemented protocol, they even recalculate checksum. Their output penetrates much deeper and tests with broader coverage.

Fuzzing is a very effective method of testing at the beginning stage of verification. First, it stresses error handling and input validation code, parts that usually is not

tested well. Second, forging specially crafted packets are actual attack mechanism of attackers. Fuzzing imitates this approach. Third, fuzzing is done with automation and hundreds of thousand of packets can be sent in a couple of days with enormous amount of permutation of errors. However, gain from fuzzing deprecates quickly; after fixing errors, fuzzing no longer finds any more errors and it just gets superficial testing same error handling code.

7.2.1.2. Stress

Some security vulnerabilities can rise (only) under stress conditions. Even in real life, attackers get more chance under stress conditions: airport security guards get distracted during peak hours and terrorists can sneak in. In stadiums, polices have to check too many people in too short of time, causing illegal items to be brought in. Software is no different from real life; although software does not get distracted, shortage of resources can cause it to malfunction.

Attacks can range from denial of service (system cannot handle any more connections and either refuses new requests or crashes) to remote code execution (some security check may not be done because of resource shortage, not verifying return values results security breaches).

There are many tools for stress testing, some of which are previewed in the next section. Every trustworthy computing project must involve stress testing during stabilization phase.

7.2.2. Tools

The purpose of this section is to make reader familiar with some of widely used tools. Tools are not covered with complete details and coverage is kept short intentionally. There will always be better tools with more features and giving more details would render this thesis out-of-date sooner. However, this section will make more developers aware of such tools and testing ideas.

7.2.2.1. Page Heap

In protected mode, memory is divided into pages and memory access is determined on page basis. Virtual memory hardware checks memory access with page size granularity. Usual size of each memory page is four kilobytes. Normally, when an

application allocates new memory from heap, heap manager reuses pages and returns pointers to unused parts of pages.

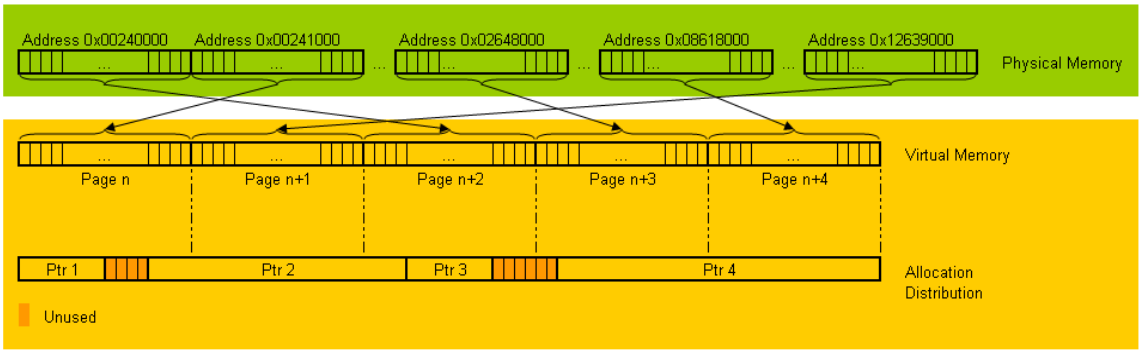


Figure 7.2: Virtual memory mapping

However, enabling page heap causes a different method to be used for more efficient error detection. New layout will be as follows.

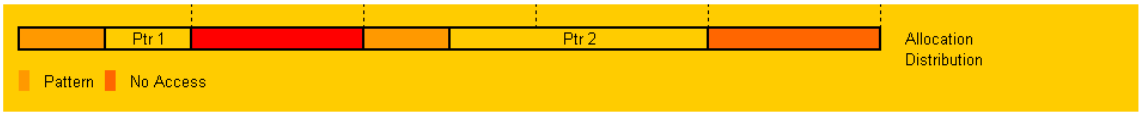


Figure 7.3: Page-heap allocation

Each memory allocation requests will be satisfied with an additional page reservation and marking it inaccessible. Allocated memory sits on the border of inaccessible page. Addresses in the page boundary before that memory are also reserved and filled with an integrity check pattern. If there is a buffer overrun in the application, it will immediately cause to access violation and throw a non-continuable exception. If there is under run, pattern before memory location will be corrupted and check during memory free will throw an exception.

Downside of page heap is that each memory allocation will cost at least 2-page size, namely 8192 bytes of allocations.

7.2.2.2. Application Verifier

This tool [11] helps developers "*identify potential application compatibility, stability, and security issues*". It includes Page Heap and many other tests, which works together with a debugger and gives detailed information about behavior of tested

application. It pinpoints potential errors and security defects like determining when an application is creating objects and assigning ACL's to that object.

7.2.2.3. Tools from SysInternals

M. Russinovich has been disassembling Windows for many years and developed great tools that are very hard to find somewhere else with those capabilities. Here, most popular of his tools are described briefly [7].

◊ FileMon

This tool monitors interaction of application with file system. Developers can use these tools to understand their applications file system behavior, to detect unnecessary file operations, to detect temporarily stored insecure sensitive information and to determine which sources their applications are using in reality.

◊ RegMon

In Windows OS, application configuration is kept in a database called "Registry". This tool monitors traffic between registry API and application. With this tool, it is possible to monitor unnecessary registry calls, to detect temporarily stored insecure sensitive information and to ensure that the application is reading correct registry keys.

◊ Process Explorer

This tool is a replacement for Windows Task Manager with providing much more information and ease of use. One advantage of Process Explorer is that it can show HANDLE information of processes; developers can check which handles are open to which sources at any given time.

7.2.2.4. Network Monitors (AKA Sniffers)

Network monitors capture traffic from wire, parse protocols to basic building blocks and display to the user. Although capabilities depend to vendors, they generally can filter out addresses and/or protocols, sort by time, length, port or addresses.

Usage of network monitors is really helpful to understand real behavior of application on the wire. Sniffers really help for speed optimizations by detecting unnecessary context switches, duplicate packets, wire errors, random check-sum

failing, resending etc. However, they are also useful for security purposes, too: it is possible to check for unencrypted sensitive data, alignment of data and timing issues. Besides, inspecting wire data gives better understanding of lower layer protocols dynamics.

While capturing network traffic, developers should be aware of switched environments. A network switch will pass traffic only for receivers on that port; it will filter out third party communication. If third party communication is important for analysis, switch port span must be used. Almost every switch with configuration interface allows this, if not, using a hub instead will solve this problem.

8. Deployment

8.1. Motivation

Shipment and installation of a program in a secure fashion is very important to ensure sustained quality of a software application. Bad deployment practices increases attack surface unnecessarily and render otherwise more robust software into a backdoor to the system.

8.2. Previous Work

Unique approach to the subject of this thesis revealed that there is less research than we think it is required. Advancements in this field are unfortunately limited with big software houses proprietary setup applications or install suites.

8.3. Minimal Setup

Bad default scripts of setup phase installs unnecessary features that will not be used by the user. Those unnecessary features increase attack surface and thus making systems vulnerable even if the system did not require that functionality in the first place. One of the examples can be default installation of IIS 5.0 with MS Windows 2000 OS. IIS had its share of security vulnerabilities. Default installation has caused almost all of the computers to be vulnerable, even if the user is, say, an accountant and does not know anything about a web server. Same thing happened with enabled RPC endpoint listener even though there is no RPC server on the system. Blaster worm has caused serious damage to the economy and personal properties [20] [21] just because of this.

Correct way of doing this is only installing absolute minimum functionality by default. Furthermore, setup frameworks should give enough information to the users to allow them to select only useful and required features for them.

8.4. Compiler Flags

Compiler documentation provides detailed information about the product. However, this section aims to make developers aware of security related switched that compilers present to the developers. Especially, this thesis is the only article, which makes developers aware of "struct member alignment vulnerability" in modern optimizing compilers.

8.4.1.1. /GS Flag

Latest versions of Microsoft C/C++ compilers provide an option called /GS. When this switch is supplied, right after function entry, compiler allocates extra space on the stack at the location before return address and fills this area with a dynamically generated (at the process entry) cookie. At the function exit, compiler also injects a small amount of code to check cookie value set at the entry. If cookie value is overwritten, so is return address probably, almost certainly due to buffer overrun. If buffer overrun is detected, a buffer overrun handler function is called (a default one is provided, however it is possible to change with "_set_security_error_handler". Default one displays a message and terminates the process.)

/GS flag is intended to reduce buffer overruns, not to completely prevent them. There can still be attacks, which can succeed despite this feature. However, this flag should be enabled on all builds, either debug or release. During verification phase, it can spot buffer overruns. During actual run time, it can prevent damage of a possible buffer overrun; attack will be turned into denial of service from privilege escalation or remote code execution.

8.4.1.2. /ZpN Switch: Struct Member Alignment

Default behavior of struct member alignment is compiler implementation dependent. Compilers are free to position struct members in the memory. They will optimize size wise or speed wise. Normally, this would not be a problem. However, if memory representation of a struct is going to be written to network, this matters.

Assume that there is network protocol, which defines a primitive remote procedure call mechanism:

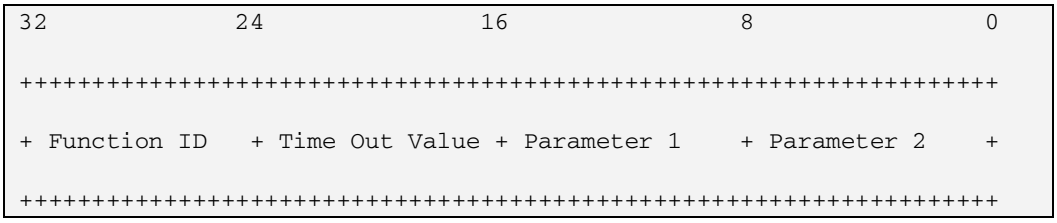


Figure 8.1: Sample network protocol

Suppose that a developer has programmed this as follows:

```
386     struct MyProto {
387             unsigned char       ucFunctionID;
388             unsigned short      usParameter;
389             unsigned char       ucTimeOut; };
390
391     void RemoteShutdown() {
392             MyProto Proto;
393
394             ZeroMemory(&Struct, sizeof(Struct));
395
396             Proto.ucFunctionID          = 0x11;
397             Proto.usParameter           = 0x2222;
398             Proto.ucTimeOut             = 0x33;
399
400             unsigned int uiSize = sizeof (Proto);
401             SendViaCurrentSocket(&Proto, uiSize); }
```

Figure 8.2: Implementation of sample network protocol

Programmer expects that this code will put HEX (11222233) to the network; however, result is compiler implementation defaults dependent. For example, compiler of Visual Studio will provide “sizeof (Proto)” as six and will send HEX (110022223300). What is happening here? Compiler aligned struct members to utilize RAM bus best.

If structure will be used as a memory template, then compiler must be tuned with switches to align struct members to one-bytes. This can be set with "/Zp1" flag in Visual Studio.

8.4.1.3. /RTCc Flag

Latest versions of Microsoft C/C++ compilers provide an option called /RTCc. When this switch is supplied, compiler injects code to check assignments against value losses. Assigning long variables to shorter ones generates an exception; where a programmer can attach a debugger and see what is going on.

8.4.1.4. /RTCs Flag

Latest versions of Microsoft C/C++ compilers provide an option called /RTCs. When this switch is supplied, compiler injects code to check frame pointers prior entering and after leaving functions. This is helpful to detect mismatched function call conventions like calling a function with cdecl although it is declared as stdcall. This flag also enables initialization of local variables to non-null values.

8.4.1.5. /RTCu Flag

Latest versions of Microsoft C/C++ compilers provide an option called /RTCu. When this switch is supplied, compiler injects code to reports when a variable is used without having been initialized. Variables that are not initialized will have random values and can result random behavior in different environments with different settings making error localization very difficult. This flag will detect them during runtime and break execution immediately by throwing an exception.

8.4.1.6. /RTCv Flag

Latest versions of Microsoft C/C++ compilers provide an option called /RTCu. When this switch is supplied, compiler injects code to report mismatched use of C++ scalar/vector new/delete operators. Scalar new should be matched with scalar delete, while vector new should be matched with vector delete; otherwise, heap corruption or memory leaks could occur. This flag will break execution by throwing an exception.

8.4.1.7. Summary

Run time check flags are very easy to use and they are very effective to localize errors. However, they are incompatible with optimizations and they make code slower. Therefore, they should be used in debugging editions of binaries.

/GS flag on the other hand, is different. Although it is essentially a run time check, it has not the limitations of other /RTC* counterparts. This flag should be used in debug and release code to prevent buffer overruns, thus to prevent taking advantage of security vulnerabilities.

8.5. Secure By Default

Applications should be deployed as secure as they can be as default. Some of the administrators may not be so knowledgeable that he can properly configure newly installed application to be secured. Moreover, configuration may be forgotten. In those cases, software will be vulnerable although it has not to be.

Checklist should contain:

- Disabling unused features,
- Creating minimum amount of server sockets,
- Disallowing guest user account by default,
- Not allowing default user accounts, especially the ones with high privileges,
- Setting up correct access control list entries,
- Doing initial configuration during setup and not delaying after setup. This ensures “secure or not installed” assertion.

8.6. Setup Package Signing

Setup packages should be signed by manufacturers to prevent tampering with packages and thus protecting customers. This will also protect reputation of manufacturer; because customers will likely blame manufacturers since all they see will be XYZ company’s application giving damage to their computer.

8.7. Removing Sensitive Data After Uninstall

Applications must be designed in a way that cryptographically erases sensitive and private data after removing the application. As always, what invisible is hard to detect and implement. Program installation is visible, and any missing file will be detected most probably because the application will not work properly. Unfortunately, if some of the important files remain, it will not be detected by the user or application.

9. Maintenance

Shipping software is generally another start of worries: Maintenance.

9.1. Motivation

Original developers start developing the next version of the product. Maintenance gets responsibility of other people that is different from original team. Moreover, companies tend to use more talented and experienced developers in main branch, rather keeping them busy with old product. After all, companies made the profit they were planning. However, bad maintenance practices can cause regressions in the code.

Regressions can cause security vulnerabilities that were not present in the original code. Proof of this can be found on security bulletins of major software houses. If looked carefully, it can easily be seen that most of the bulletins refer to only some version of a product. For instance, [45] is a security bulletin from Microsoft Corporation that addresses a security vulnerability that affects only a subset of operating systems. Obviously, this defect is introduced with a regression.

Motivation for working on that phase comes from the fact that sustained quality engineering is as much as valuable as developing the application.

9.2. Previous Work

Software engineering science has studied maintenance phase thoroughly over the years.

9.3. Regressions

Every change in the code can have effects locally or globally. Local changes are generally expected and intended. However, global effects are rarely well understood or intended. Such side effects occur especially in the cases where several people

maintain a project with large code base that is not well modularized into independent parts.

Regression is defined as new defects in the code base after changing code; new erroneous behavior will be present and software will fail tests, which it was passing before. Regressions can cause very bad security vulnerabilities. Unlike normal program features, real security of software is not visible, and it is always hard to test invisible. Security of rather large software is impossible to prove; only absence of security can be proven with a successful attack. Therefore, a regression passing all security and non-security tests can still be vulnerable to yet unknown attacks.

9.3.1. Research on the Effects of Regressions

In order to demonstrate effects of regressions, security bulletins are researched for indications of regression origin.

9.3.1.1. Method

For this purpose, all security bulletins since 2000 from Microsoft Corporation are examined. Regression indication is decided as follows:

If there is vulnerability, which only effects serviced products, or

If there is vulnerability, which only effects new version of the product but there has been no DCR.

9.3.1.2. Results

"Table 9.1: Security Improvement Research of Microsoft Corporation" presents the results of this research.

Until 2003, Microsoft Corporation did not encounter high percentage of regression bugs. Even in 2003, affected software was mainly down-level platforms. With the increasing number of supported platforms and introduced changes to code base, regression bug counts started increasing. Percentage is increased from only 3% to 28%.

Table 9.1: Security Improvement Research of Microsoft Corporation

Years	Total Bulletin	Indicated Regression
2000	100	**3**
2001	100	**6**
2002	72	**5**
2003	51	**4**
2004	**44**	**12**

Another interesting result is that trustworthy computing initiative in 2001 of Microsoft Corporation has proved to be very effective. After 2001, count of security bulletins is drastically reduced.

9.3.2. Regressions during Bug Fixes

The biggest problem with bug fixes is regressions. A regression occurs when code change has unexpected side effects in other parts of code causing code defects. Since it is generally hard to tell where the next regression will show up, they are easy to miss during tests.

Also should be noted that Systems suffering from high level of regressions are generally poorly designed system with high amount of cross dependencies. A developer unfamiliar with the project (old developer who has forgotten minor details of project or a new one stranger to the project) will have hard time in such systems. Therefore, fix triage should be made very carefully and selectively.

9.3.2.1. Perfective

Perfective bug fixes are the fixes that do not fix known or important issues. Developer thinks that that fix could be nice to have it and changes code. Although this type of changes can be acceptable at the early phase of development, it should be avoided as the code base matures. They can be exercised on service pack branch (service packs are really well tested), but they should be kept away from day to day

customer fixes (which normally has limited test coverage). It should be kept in mind that every change in the code can cause regressions somewhere else and decision should be made if the fix is worth of taking that risk.

9.3.2.2. Corrective

Corrective fixes are compulsory fixes due to customer requests, publicly known vulnerabilities or blocking issues. Since they must be performed, there is not an option for manager but accepting that requests. However, developers should resist the tendency of perfective fixes while working on a necessary one. First, this action will increase the chance of regression because that kind of "minor" perfective changes will be most likely less understood. Besides, it will make code reviews less effective and confusing. Second, porting this fix to another platforms or versions will be more complicated.

9.3.2.3. Final Words

An anonymous saying can summarize this section very good:

> *"If not broken, don't fix it"*

9.3.3. Detection: Code Reviews

Every change should be checked with static code analysis tools and then send to a peer for code change review. According to the experiences of us, another look that is free of presumptions can be very effective while detecting errors.

Review should be done by someone who has been involved in the project for long term and has a good deal of overall knowledge related to the project, especially about inter-module dependencies.

A reviewer should look, besides other domain specific needs, for design of patch, correctness of patch, obvious and possible side effects of code change, implementation details, usefulness, fit within its module, fit within application globally, usage of API's, banned API, commenting, and style. Furthermore, reviewer should verify changes for conformance to company policy and project handbook.

9.3.4. Prevention: Bug Fix Check-Ins

Each code check-in should fix exactly one bug, not anymore. This discipline will help tracing regressions back to the code change, which introduced that defect and

therefore making analysis of bug shorter. Keeping fix times shorter means high responsiveness to publicized attacks and keeping attack windows small.

9.3.5. Prevention: Keeping Complexity Down during Implementation

Regressions occur mostly because of high complexity of source code. Relatively less experienced developers are not familiar with the code and complex code base does not make their job any easier. Following simplicity techniques that were mentioned in Design and Implementation chapters helps to keep complexity at a manageable level.

9.4. Design Change Request's (DCR)

Design of a product is done at the beginning of a project. After product is at the market, DCR is not a very welcome request. It has many threats in it:

Difficult. It is very difficult to change an implementation written for another design. There will be many contracts among objects of the design. The requested change will probably break these contracts. Tracing references to contracts are very time consuming; in fact, covering every single one of them is almost impossible in a large-scale project. These changes will be very expensive and time consuming, for both development and verification.

Decorator design pattern provides a method to add required features later.

Error prone. Difficult jobs are error prone by their nature, and this is not an exception. Broken contracts will damage the structure of software. Compiling without any errors mean less, since contracts are not only interface base, but also behavior base. At the very least, owner ship of objects (pointers, handles, etc) is very important for a robust system; possibly broken contracts will make it very difficult to track ownership, double-free issues will begin to show up.

Hard-to-test. Changed design and code base will require new test cases, which will take considerable time. Companies will not likely want to spend high amount of money and will rush to ship change. However, lack of test will result in errors, possibly security vulnerabilities.

DCR's should be listed for next major version of product when there will be large redesign / code change and long verification time.

10. Examination of Existing Vulnerabilities

10.1. Motivation

Author has given information about several methods to prevent security vulnerabilities throughout the thesis. In this chapter, goal is demonstrating effectiveness of the suggested methods. This has two advantages. First, reader can understand value of methods. Second, reader can relate suggested methods to real world problems.

10.2. Approach to Subject

To demonstrate effectiveness of methods, 18 of sample real life security vulnerabilities will be examined.

10.3. Examples from Real Life

10.3.1. MS00-001 "Malformed IMAP Request" Vulnerability

This Description is taken from Microsoft Corporation TechNet web site http://www.microsoft.com/technet/security/Bulletin/MS00-001.mspx

> *The IMAP service included in MCIS Mail has an unchecked buffer. If a malformed request containing random data were passed to the service, it could cause the web publishing, IMAP, SMTP, LDAP and other services to crash. If the malformed request contained specially crafted data, it could also be used to run arbitrary code on the server via a classic buffer overrun attack.*

Since we cannot reveal specifics about this code defect, discussion will be limited to overall buffer overrun aspect.

Author has suggested usage of COM modules in risky parsing environments. IMAP can be considered risky because e-mail protocols were established a long time ago and therefore they are loosely formatted, making parsing a difficult job. If Microsoft Corporation had used COM modules to separate risky portion, arbitrary code

execution would not be a real security threat, because process would be running in a restricted environment. Attacker would be able to do only what the process is allowed to do, probably nothing but parsing input data.

Although it was not available at those days, Microsoft Corporation C/C++ compiler /GS flag could potentially prevent a buffer-overrun vulnerability.

10.3.2. MS00-005 "Malformed RTF Control Word" Vulnerability

This Description is taken from Microsoft Corporation TechNet web site

http://www.microsoft.com/technet/security/Bulletin/MS00-005.mspx

> *RTF files consist of text and control information. The control information is specified via directives called control words. The default RTF reader that ships as part of many Windows platforms has an unchecked buffer in the portion of the reader that parses control words. If an RTF file contains a specially-malformed control word, it could cause the application to crash.*

This bug could have been prevented with following principle “tight tunnel”. RTF parser apparently is not checking its input correctly and this results invalid entries to find its way deep into the code. This bug is not a regular buffer overrun attack, because read data is not directly written to the buffer, but after some initial processing (Otherwise it would be a full-blown security vulnerability allowing attacker to run code in host process context). RTF parser should have checked validity of input and verify that it is indeed one of the valid control commands.

10.3.3. Driver-Monitor Framework Unitialized Out Parameter Vulnerability

This vulnerability has been present in one of the projects that the author of this thesis has developed.

```
402     void ReadUserName(
403         OUT szUSerName[32]) {
404
405         //Zero out parameters..
406         ZeroMemory(szUserName, sizeof(szUserName));
407         ... }
```

Figure 10.1: Driver Monitor Framework Vulnerability

This vulnerability is caused because of improper size determination during zeroing out parameters. Sizeof(szUserName) statement returns whatever the size of regular pointer is, which is 4 in 32 bit platforms. Of course, the result is shorter than the real size of the parameter and therefore parameter ended up without being zeroed completely. Since this parameter was reused in other places, this has caused a security vulnerability, though hard to discover one. Fortunately, this vulnerability has not been exploited.

10.3.4. Linux Kernel Backdoor Attempt

There has been a attempt to plant a backdoor into the Linux kernel in 2003. Although this is not a security vulnerability in the sense of code defect, it could still be caught if one line code mistake catalog have been used as a checklist. Vulnerable code is below:

```
408                     ..
409                     ..
410                     schedule();
411                     goto repeat;
412             }
413             if ((options == (__WCLONE|__WALL)) && (current->uid =
0))
414                             retval = -EINVAL;
415             retval = -ECHILD;
416     end_wait4:
417             current->state = TASK_RUNNING;
418             ..
419             ..
```

Figure 10.2: Linux Kernel Backdoor Attempt Source Code

What seems as an innocent comparison statement actually sets user id to zero, which is the id of the most privileged user account in Unix world. Hacker first checks to see if some special flags have been set, then set its user id. Since this is kernel code, modifying user tokens is that easy. A checklist of one-line code mistakes, or an automated tool which consumes that list could detect this defect right away.

However, Linux kernel source depot did not have such a sophisticated mechanism at that time and a human caught this error.

10.3.5. Apache Web Server Chunk Handling Vulnerability

Apache is a popular platform-independent HTTP 1.1 compliant web server that is mostly used in Unix environments, most notably Linux. There has been a vulnerability in 2002, description of vulnerability can be seen below [55]:

> *Versions of the Apache web server up to and including 1.3.24 and 2.0 up to and including 2.0.36 contain a bug in the routines which deal with invalid requests which are encoded using chunked encoding. This bug can be triggered remotely by sending a carefully crafted invalid request. This functionality is enabled by default.*

[54] is CERT report about this vulnerability. What is important about this defect is that it affects many versions of many products.

Even in open source vulnerability reports, code defects are not mentioned with a good clarity to prevent encouraging attackers. Following examination is done by finding out defective and fixed versions, windiff'in them and inspecting source code.

Old function to determine the chunk size:

```
420     static long get_chunk_size(char *b)
421     {
422         long chunksize = 0;
423         while (ap_isxdigit(*b)) {
424             int xvalue = 0;
425
426            /* This works even on EBCDIC. */
427             if (*b >= '0' && *b <= '9')
428                 xvalue = *b - '0';
429             else if (*b >= 'A' && *b <= 'F')
430                 xvalue = *b - 'A' + 0xa;
431             else if (*b >= 'a' && *b <= 'f')
432                 xvalue = *b - 'a' + 0xa;
433
434             chunksize = (chunksize << 4) | xvalue;
435             ++b;
436         }
437
438         return chunksize;
439     }
```

Figure 10.3: Appache Vulnerability: Old Code

New function for the same purpose can be seen at the next page. Besides the bug itself, we first want to note other improvements. Since this code has caused vulnerability, the programmer, who fixes the source code took additional countermeasures instead of just fixing the defect.

First, curly braces are used to encapsulate even single line if-statements. This is a good coding style because it prevents accidental under-coverage or over-coverage. Second, Api export functions are noted with “ap_” prefix. Although we would suggest using namespaces, since the source file is C, prefixing function names are all

we have. Third, now there are spaces between the local variable definition and while loop.

```
440     API_EXPORT(long) ap_get_chunk_size(char *b)
441     {
442         long chunksize = 0;
443         long chunkbits = sizeof(long) * 8;
444
445         /* Skip leading zeros */
446         while (*b == '0') {
447             ++b;
448         }
449
450         while (ap_isxdigit(*b) && (chunkbits > 0)) {
451             int xvalue = 0;
452
453             if (*b >= '0' && *b <= '9') {
454                 xvalue = *b - '0';
455             }
456             else if (*b >= 'A' && *b <= 'F') {
457                 xvalue = *b - 'A' + 0xa;
458             }
459             else if (*b >= 'a' && *b <= 'f') {
460                 xvalue = *b - 'a' + 0xa;
461             }
462
463             chunksize = (chunksize << 4) | xvalue;
464             chunkbits -= 4;
465             ++b;
466         }
467         if (ap_isxdigit(*b) && (chunkbits <= 0)) {
468             /* overflow */
469             return -1;
470         }
471
```

```
472         return chunksize;

473     }
```

Figure 10.4: Apache Vulnerability: New Code

The bug itself could have been prevented by using remoting mechanisms that was mentioned in design phase. In Linux, there is no COM technology, but underlying structure is anyway there: Rpc. Since Rpc is a very efficient mechanism of inter-process communication, its usage affects performance only marginally and linearly. HTTP parser could be in a non-privileged process. It would parse the requests and create an internal representation of parsed request. Then, privileged peer could process the request and perform tasks that require privilege.

10.3.6. Apache Environment Expansion Vulnerability

This code defect is in configuration file parsing logic. There is buffer overflow vulnerability if a specially crafted input file is passed to the Apache web server. This vulnerability is difficult to exploit, attacker must be local user and manage to command to server to parse his or her configuration file. Since the fixed code is long and involved, it is not included in this thesis. However, interested readers are encouraged to see the source code ".\server\util.c" and compare the versions 2.0.50 and 2.0.51. Old version can be seen below:

```
474     AP_DECLARE(const char *) ap_resolve_env(
475                                  apr_pool_t *p, const char * word)
476     {
477             char tmp[ MAX_STRING_LEN ];
478             const char *s, *e;
479             tmp[0] = '\0';
480
481             if (!(s=ap_strchr_c(word,'$')))
482              return word;
483             do {
484                     /* XXX - relies on strncat() to add '\0'
485                      */
486                     strncat(tmp,word,s - word);
487                     if ((s[1] == '{') && (e=ap_strchr_c(s,'}'))) {
488                             const char *e2 = e;
489                             char *var;
490                             word = e + 1;
491                             var = apr_pstrndup(p, s+2, e2-(s+2));
492                             e = getenv(var);
493                             if (e) {
494                                 strcat(tmp,e);
495                             } else {
496                                 strncat(tmp, s, e2-s);
497                                 strcat(tmp,"}");
498                             }
499                     } else {
500                             /* ignore invalid strings */
501                             word = s+1;
502                             strcat(tmp,"$");
503                     };
504             } while ((s=ap_strchr_c(word,'$')));
505             strcat(tmp,word);
```

```
506
507             return apr_pstrdup(p,tmp);
508     }
```

Figure 10.5: Apache Vulnerability: Environment String Expansion

Although this defect is unlikely exploitable, it is included here nonetheless. What we consider as biggest mistake here is not the bug itself, rather the decision to use a proprietary environment expansion functions while there are plenty of implementation there. Linux kernel, most notably, has the highest quality examples of such routines. Programmer not only develops its own version, but also develops it with a known bad API such as 'strcat". If readers see the source code, they can be stunned because of the buffer overflow possibilities with such a recent release of Apache web server.

10.3.7. Tacacs+ Server Vulnerability

Tacacs+ is a network authentication, authorization and accounting protocol that is used especially in Cisco based networks. The author of this thesis has developed a version of Tacacs+ server in 2003. There was security vulnerability in the code that existed because of inconsistent usage of parameters among functions. Luckily, that piece of code was eliminated during testing. Problem is that some type of functions accepted timeout in milliseconds, while other functions accepted it in seconds. This could cause much longer waits in server code, which block threads. After a while, server would starve of threads and attacker could have managed denial of service attack. Clearly, this is against tight tunnel principles. Data must flow in the narrowest path possible. With different representation, it is given chance to flow in a broader channel.

10.3.8. Vulnerability in MS Message Queuing

Message queuing is used to enable inter platform communication over well-standardized API's. Differently than other technologies, it also supports guaranteed message delivery. This technology is mostly used in high-end critical servers. There is a publicly known vulnerability in Microsoft Corporation's implementation of that technology. It is documented in [57] on April 14, 2005. Vulnerability in this

component is a buffer overrun that is caused by an unchecked buffer. An attacker, who exploits this vulnerability, can execute arbitrary code remotely of his or her choice. What makes this vulnerability critical (In our opinion. This is supported by public announcements of Microsoft Corporation in [57]) is that the injected code can do everything that an administrator can do locally on system console. This is possible because of the user account the MSMQ service is running in. In our opinion, biggest mistake here is putting a network service in SYSTEM account, the most powerful account in Windows environment. If high privileges were required, COM compartmentation could be used to establish boundary layer between modules with different privilege requirements.

10.3.9. Rpc Blaster Worm

Although this worm as known Rpc worm, the truth is not exactly that. Vulnerability existed in a Dll, which is called RpcSs.Dll. This Dll is responsible of MS COM technology implementation and Rpc component has only one feature there: Endpoint mapper. Endpoint mapper is helper functionality for Rpc; it resolves server endpoints from UUID's.

A buffer-overrun vulnerability is exploited in August of 2003 [60]. This worm is another instance where the damage is increased due to high privileges of the attacked process. Mitigation of restricted user account would make this worm less effective. Cost for each large enterprise is estimated over $7 million. Blaster worm could do much more than it was originally doing, which is performing DoS attack to WindowsUpdate.Com, public patch download site of Microsoft Corporation. Since the RpcSs runs in system context, possibilities are limitless. Even an unsuccessful attempt to inject code would most likely kill RpcSs process, which is critical system process and causes system to initiate shutdown.

10.3.10. Traffic Analysis Vulnerability in SafeWeb

SafeWeb is a public internet access proxy that is used to browse the web disguised. It has been used popularly in countries (like China), which prohibits free browsing of Internet. There is a publicly documented vulnerability in the product [59]. An extensive traffic analysis can reveal which IP address is visiting which site. Since IP addresses can be traced back to real persons, this causes serious functional deficiency. Attack is performed by recording network traffic to and from SafeWeb

servers. Than, each request is fingerprinted in the number of packets it contains and the sizes of packets. Future requests are compared with fingerprint database to understand who is connecting where. What is interesting about this vulnerability is that it can be a real threat to actual human life, since some of customers of SafeWeb reside in countries, which has extensive amount of death penalties.

10.3.11. MS SQL Server 2000 Slammer Worm

What makes this worm and Blaster worm so harmful and effective is that they do not require any kind of user interaction and they inject code to the processes those are running in highly privileged context. MS SQL worm is a vulnerability that is caused by a buffer overrun.

10.3.12. Vulnerability in the License Logging Service

License logging server is a component of Microsoft Windows Operating System and is used in Client Access License environments for backup purposes. There is buffer-overrun vulnerability that is publicly known [62]. If successfully exploited, an attacker can take complete control of the system. What makes this case particularly interesting is the impact of vulnerabilities among systems. Although this component is almost in every major Windows version, latest Windows version as the time of this writing is considered not critically impacted. Cause of this is that the service is not enabled by default in Windows 2003 and only an administrator can enable this service. Although Windows 2003 has the same defective code, most of users of this version is not affected. This case is good example reducing attack surface practice. Following, excerpt from [62] can be seen.

Vulnerability Identifiers	Windows NT 4.0	Windows 2000 Server Service Pack 3	Windows 2000 Server Service Pack 4	Windows Server 2003
License Logging Service Vulnerability - CAN-2005-0050	Critical	Critical	Important	Moderate

Figure 10.6: Impact Difference among Different Versions of Windows OS

The reason why Windows 2000 Service pack 4 is not critically affected is that server hardening guide for that particular version suggested disabling the service, although it was enabled by default.

10.3.13. Named Pipe Vulnerability

Named pipes are a transport layer protocol that is used in Windows environment frequently. There is a publicly known vulnerability in that functionality in Windows XpSp1 and XpSp2 [64]. This case is another example of reducing attack surface. Computer browser service is essential in order to exploit this vulnerability, but is disabled by default in Windows Xp service pack 2.

Vulnerability Identifiers	Windows XP Service Pack 1	Windows XP Service Pack 2
Named Pipe Vulnerability - CAN-2005-0051	Important	Moderate

Figure 10.7: Impact Difference Among Different Versions of Windows OS (2)

10.3.14. Vulnerability in PNG Processing

PNG is a portable graphic format that is widely used in Internet environment to share image files. Microsoft Windows Operating System has a publicly known vulnerability that is caused because of integer overflow while parsing input data [63]. An attacker can execute remote code by successfully exploiting the vulnerability. File parser vulnerabilities are especially dangerous because it is not the same kind of viruses that people are generally aware of. People expect harm from executables, batch files, and macros and even from scripts. However, users do not expect to be infected while browsing a web site that consists of just plain HTML and some images. This kind of attack is so effective that a system can be affected just because the user displayed an image. This and other similar vulnerabilities give several lessons:

- An input should never be trusted
- All input can cause vulnerabilities, even if the parser does not deal with networking or executable files.

- Common system components must have very high security standards since these components are being used by all users of the system. Moreover, they cannot be disabled or separated from the system.
- An attack may come from least expected sources and may still have such an astounding audience.

10.3.15. GDI+ Vulnerability

A very similar and probably wider spread (because JPEG is more popular) attack was GDI+ vulnerability that occurred in 2004 [64]. This attack was more harmful than PNG because vulnerability existed in more than one product. Users had to go to official Windows update site. During visit there, a scanning tool was deployed to detect application that uses defective GDI+ libraries. However, that scanner could not scan all applications either; it was only compatible with products of Microsoft Corporation.

10.3.16. Apache 2.0.49 64-Bit Vulnerability in Mime Parsing Code

Since 64-Bit address, space and number limits are much larger than their 32-Bit counterparts are they increase visibility of code defects. One of such defect was present in Apache web server, which caused heap based overflow if successfully exploited.

Complete source function will not be included here, since it is too long. However, added fix will be presented here.

```
509     if ((fold_len - 1) > r->server->limit_req_fieldsize) {
510           r->status = HTTP_BAD_REQUEST;
511           /* report what we have accumulated so far before the
512           * overflow (last_field) as the field with the problem
513           */
514           apr_table_setn(r->notes, "error-notes",
515                 apr_pstrcat(r->pool,
516                       "Size of a request header field "
517                       "after folding "
518                       "exceeds server limit.<br />\n"
519                       "<pre>\n",
520                       ap_escape_html(r->pool, last_field),
521                       "</pre>\n", NULL));
522           return;
523     }
```

Figure 10.8: Apache 64 Bit Vulnerability Code Patch

Critical line is line number 509. This if statement ensures that current fragment length is not longer than the limits. In the absence of this limit, server could survive attacks in 32-bit platforms, since numeric range is much lower. However, with 64 bits huge numbers, visibility of the defect increases. Therefore, we defended the importance of 64-Bit verification in our work.

10.3.17. Linux Real Time Clock Vulnerability

Real time clock functionality does not initialize their structures in Linux Kernel 2.4.23 and earlier. This results in kernel data leak to user space [66]. We consider this defect because of not following tight tunnel principles correctly. Although a tight tunnel principle mostly defines runtime behavior, it should be noted that proper initialization is also an important aspect of tight tunnel principle.

10.3.18. Final Bug: Ping of Death

Ping of death is one of the oldest and yet most effective attacks. Attack basically consists of a ping packet that is longer than legally allowed size, which is 2^{16}-1=65535 bytes. This is possible because of fragmentation logic of IP packets. Receiving computer is attacked when assembling packet fragments into a single buffer. It makes sense to have a fixed size buffer for IP packets, because there is an absolute limit in the size of packets. Therefore, many implementations had a buffer size of 16 bits. Overwriting this buffer caused different effects on different systems.

The interesting thing about ping of death is the low quality of TCP/IP stacks just 10 years ago. Imagine that an attacker can crash any computer with just a ping packet. That is an awesome power, attacker does not need to know anything but the IP address. We added this bug here to demonstrate how far the computer technology is advanced in means of security in the last decade.

10.4. Which Failures and Defects Are More Critical

This question has been discussed for a long time among academic environments and no one seems to have a correct and only answer. Reason for that is there is a huge problem in the security assurance of software products: In theory, even the smallest amount of defects can cause serious security vulnerability. In addition, this defect does not have to be in the design or deployment phase either. Any of the one-line code defects can open the doors for remote code execution attacks. This is very unlikely to other engineering disciplines that we are used to. For instance, nobody would expect a bridge collapsing because of a single forgotten bolt. However, very unfortunate for software, delicacy of code harmony makes software very fragile. Important thing is in which context the program is running in and the location of defect.

One argument is that some sorts of defects are more important compared to others are. If thought superficially, this sounds right, because most of the security bulletins talk about buffer overruns after all. Therefore, can we say that unchecked buffers are most critical errors? Not quite likely, we think; a very important issue is omitted here: An integer overflow can cause a buffer overrun, too. Unintentional assignment operator in an if-statement can cause integer overflow, too. In addition, bad style can

cause unintentional assignment operator in the if-statement. Finally, bad design spec that does not dictate a style can cause bad styling. Everything is in a fragile harmony, and therefore software security consists of total quality.

We believe that any categorization of defects and mistakes should be taken as rough guidelines only and should not be considered as a serious reference. Since code bases are very large, bad surprises are happening always. With sensitive building blocks upon each other, software becomes as a house build from playing cards, each piece is fundamental.

10.5. Security Push Practices

Although we cannot comment about criticality levels of mistakes, we can suggest where to start for any security push. This is because some practices proved to be more beneficial in the short run. This guideline is especially useful if software is suffering vulnerabilities and there is only short amount of time to fix them.

A security push can be done only if everybody in the team believes its necessity and spares time for just security inspections. This was the way that Microsoft Corporation conducted Trustworthy Computing Initiative since 2001 and related security pushes in 2001 (.Net Framework) and 2002 (Windows Server) [59].

10.5.1. Consider Reducing Attack Surface

Obviously, if an application is suffering from much vulnerability, first thing to do should be reducing the number of vulnerabilities. Easiest way to reduce the number is cutting of the features. Generally, if the future is not present, an attacker cannot attack to it. However, this is not always practical or even possible. What can be done alternatively is that reducing the number of services that is enabled by default. There are several advantages of this, as documented in [59]: Attackers get more reluctant to attack that feature since less people is using it. Moreover, even if the vulnerability is exploited, less people get affected. In addition, people who do not deploy that feature can wait for installing patches until it is most appropriate time for them.

We understand that some applications can be very complex to make such changes during maintenance phase. Another solution for these products can be implementing

a custom designed filter application. After filtering harmful or dangerous content, application can continue processing pre-processed data.

10.5.2. Consider Alternative Designs

If the program already consists of visible and clearly defined modules, implementing boundary access checks can be easy to implement, of course depending on the application. One example is usage of Com modules and implementing LUA principles. This type of change should not require any extensive amount of modifications in the actual program logic.

10.5.3. Consider Using Automated Tools

One-line code mistakes can cause much more trouble than their apparent sizes, as discussed throughout the thesis. Discovering such defect can be done with code reviews. However, in order to have effective code reviews, different people than the implementer should read the code, of course with the presence of implementer. Gathering all of these people and spending time on reviews may not be feasible always. In that case, automated tools can be used to increase confidence in the code base. The tools that are mentioned in this thesis can be a good point to start.

Another type of automated tool is the compiler itself. Some compilers present nice features to detect and prevent some kind of vulnerabilities, most notably buffer-overrun vulnerabilities. Considering alternative compilers can result in better tool selection that helps improved robustness.

10.5.4. Consider Being Proactive in Finding Vulnerabilities

Fixing exploited bugs is not that good as fixing them before they actually become vulnerabilities. Therefore, being proactive can really help in improving quality before it gives damage, to consumers or to producers. Moreover, being proactive in finding vulnerabilities helps for proactive preventions.

Best proactive method is considered usage of intelligent fuzzers. What a fuzzer does is changing actual input data from what it is, and converting it to an illegal shape that is unexpected by the input parser. After all, this is what attackers do to discover vulnerabilities. Intelligent fuzzers are discriminated from standard fuzzers by their knowledge of input format. Changing random bits in the input is not such a good

idea in most cases, because it usually breaks simple checks like CRC checks. Packets are discarded before penetrating deep enough to cause access violations. What a smart implementation can do is changing critical fields (length indicators, timestamps, and type indicators), recalculating CRC or any other temper proof mechanism.

Security makes implementations of fuzzers difficult. Since encrypted packages are no longer easily modifiable in a smart way, internal hooks can be used to inject fuzzer between output processors and encryption logic.

10.6. Checklist for the Covered Topics in this Thesis

This section aims to provide a checklist about the topics covered in this thesis. This checklist should not be considered as a complete checklist to write vulnerability free programs. Writing such a checklist is impossible, indeed. However, this checklist summarizes new ideas of this thesis and some other ideas that are used as background information with the goal of gathering them into a place where a designer, programmer or servicing staff can use a reference to ensure coverage at least of the topics of this thesis.

Table 10.9: Checklist for the Covered Topics in this Thesis

Checklist Item	**Y**	**N**
Am I sure that I have understood my potential enemies?		
Am I sure that I have understood possible motivations of my enemies?		
Do I have a clear understanding of why my system may be susceptible to attacks now or in the future?		
Am I sure that I have understood attack types?		
Do I have an understanding of roles of modules of my application in the sense if they are client, server or both?		
Does my client know exact requirements of this project?		

Do I know the exact requirements of this project?		
Does the client know exact security requirements of this project?		
Do I know the exact security requirements of this project?		
Am I fairly sure that the requirements will not change in this version of the product (since it would cause serious design changes, therefore code changes and finally security vulnerabilities)?		
Am I sure that all of the requirements are necessary for the success of the project? Moreover, am I sure that deducting any features from the set may cause decrease in usefulness?		
Do I know what the effect of the worst attack is?		
Do I know associated costs with DoS, DDoS, Privacy Compromise, Remote Code Execution?		
Do I know what kind of private data the application can lose at most?		
Do I know what kind of security-usability-performance-cost tradeoffs I am making?		
Am I sure that I have spoken security trade-offs with my client and made them understand what the compromises are? Do they have clear understanding?		
Do I know design patterns?		
Do I know security implications of design patterns?		
Am I proficient enough to select among design patterns according to my design and security requirements?		
Do I know in what kind of network environment will be my application running in (Strictly DMZ in access controlled systems room, strictly DMZ, strictly intranet, intranet, and internet)?		
Do I know if my application will be susceptible to traffic analysis, packet repeat or any other below L4 alterations?		

Do I know what the encryption modes are?		
Am I sure I made correct encryption mode selection for my project?		
Am I sure that I made correct implementation of the encryption mode I have selected for my project?		
Did I inspect network traces to ensure that my program is indeed running in the desired encryption mode?		
Do I make compression before encryption?		
Do I know how a buffer-overrun-attack works?		
Did I select correct user account for the applications runtime context?		
Am I sure that the selected user account is minimum privileged account that is possible for the correct operation of the application?		
If the user context is privileged, did I modularize my application into parts that requires different levels of privileges and access rights? Moreover, did I pay attention to the different roles of modules (like accepting input, parsing, performing requests, logging, connecting other services)		
Do I know infrastructures of Rpc or related technologies (Java RMI etc)?		
Do I know COM or related technology (CORBA, SOAP etc.) infrastructure?		
Is there a way that I can modularize my application into modules that each module runs with minimum privileges that is required for that module?		
Am I sure that overhead of Rpc or Com usage is acceptable according to requirements and security trade-off decisions?		
Am I aware of attack surfaces and threats that are posed to my application?		
Did I obey the checklist in Tight Tunnel section while designing interfaces and implementing the code?		
Am I familiar with one-line code mistakes?		
Did I write my code keeping possible one-line code mistakes in mind?		

Did I check my code against one-line code mistakes manually?		
Did I check my code against one-line code mistakes automatically?		
Am I sure that I have used all available automatic code analysis tools?		
Is usage of automatic code analysis tools a sign-off requirement?		
Do I know what exception safety is?		
Do I know how to establish exception safety?		
Am I verifying implementation against exception safety requirements in each milestone?		
Do I understand importance of consisting coding style and format?		
Do I have a consisting code style and formatting among all source files and headers? Moreover, are these practices documented in specifications so that future maintenance can adhere to the style?		
Am I aware of user mode and kernel mode concepts?		
Do I know what can be done wrong with kernel mode access checks?		
Am I doing kernel mode access checks properly?		
Do I have a specification about the relationships among functions?		
Are my functions doing only one thing and doing it well?		
Do I reuse functions so that: I will not write defective code while rewriting same functionality several times / my changes are consisted / verification time decreases as amount of code decreases?		
Am I aware of tools that are at my disposal?		
Did I speak with my colleagues about the tools that they are aware of?		
Am I sure that I am using correct tools?		
Am I aware of software libraries that are at my disposal?		
Do I use them according their motivations and expected usage patterns in my projects?		

Do I know my applications thread model?		
Will this thread model be consistent during lifetime of processes? Will this thread model be same in future releases?		
Am I effectively using libraries?		
Will we support 64 bit? Did I check that my application is designed to correctly support 64-bit platforms?		
Am I deploying the application with minimum set of by-default enabled features?		
Do I know my compiler and its capabilities well?		
Did I compare my compiler to other compilers before making my decision to use it?		
Do I effectively use compiler flags to prevent vulnerabilities?		
Is my deployment secure by default? (No administrator action is required to make it more secure, like setting registry settings or enabling/disabling switches)		
Do we sign our releases so that our consumers can be sure that packages are indeed coming from us?		
Are our uninstall functionalities working properly? Did we test it? (If custom setup tools are used, their verification is not done most of the time)		
Are we aware of any sensitive data that we are keeping on disc or on any other type of permanent storage?		
Do we have a solid maintenance process?		
Do I know what a regression is?		
Do I believe that me or any other people involved in maintenance process knows regressions and has experience in preventing them?		

Do we have a solid bug bar to be used while triaging bug fix requests? (Excessive amount of modification causes regressions, that can cause vulnerabilities)		
Are code reviews a standard practice in our maintenance processes?		
Is our developers' proficiency enough (have a good understanding of the code, at least six months of experience) with the code? If not, are we enforcing architect review before checking in the changes?		

11. Conclusion and Final Words

This thesis has examined all phases of waterfall methodology from security point of view. Requirement analysis phase proved to be elaborated and we did not have anything to add. However, design, implementation, verification, deployment and maintenance phases presented areas for improvement and we analyzed them in this thesis. This thesis is a valuable resource for further advancements in secure application development.

11.1. Results

Security traits and robustness of software can be increased in huge amount with correct decision in design phase. Similarly, a bad design can render otherwise good security tools useless. Even worse, bad design can cause to false sense of security, which is only good for attackers. Even the oldest technologies can present new usage platforms if they have sound design. Design patterns research showed us that good security is another aspect of good design and must be taken account always; usage of design patterns may support a good structural design, but it cannot guarantee secure design. Secure design is another subject that academics are still working on.

Researches in implementation phase showed us that even smallest part of code could cause big vulnerabilities. Furthermore, it proved usage of static code analyzers to prevent detectable code defects. A research about formatting and commenting of source code revealed that even high caliber developers tend to omit these good habits, yet they want to have them. That survey also revealed that there is high benefit with keeping code readable.

Verification phase of software lifecycle is supported with many tools; most useful of them is examined in this thesis. Apparently, many software companies are not using them effectively, because they could have prevented many of the vulnerabilities. We consider this fact as a result that demonstrates unstructured approach to security practices.

Deployment phase proved to be least researched area of waterfall methodology. Discovery of this fact is probably because of the unique approach of this thesis. Contribution of this thesis is examination of compiler switches from security point of view.

This thesis put the sometimes-overlooked importance of maintenance phase in front of the eyes of researchers. A research of Microsoft Corporation security bulletins revealed that even software a giant like Microsoft Corporation is suffering from regressions. That research also showed that security initiatives are really helping Microsoft Corporation to reduce the number of successful attacks and to increase company reputation. This thesis described important points to prevent regressions from happening.

This thesis also presented a checklist to ensure coverage of new topics that are unique to this thesis. Furthermore, there was a chapter about retrofitting existing applications in a time-limited environment. This structured approach has been very helpful in authors personal projects' security pushes.

11.2. Further Research Areas

Security of applications is still fertile and fruitful area of computer science. Further researches in that area will contribute in trustworthy computing environments, which we believe is one of the main building blocks of advanced integrated computing systems. Although each technologic advancement has its own place, we still think that without proper security countermeasures, they will be rendered practically useless.

One of the biggest advancements in security point of view could be done with articles that present information about advanced code analyzing tools. There are some tools in the public domain, but they just analyze code statically. Static analyze means statistically analyzing without actually executing and further interpreting the code. Dynamic analyzing methods can help defining security vulnerabilities and understanding possible weakest links. They can especially contribute in society by analyzing nested code, code relation, data flow, binary representation and execution flow. These areas are weakest areas of static analyzers.

Another code execution analyzer advancements can be in variable usage area. Each variable in each scope becomes different values throughout its lifetime. Some of them can reach or exceed bit limits of values. Even approaching to limits can be alarming. A smart analyzer should understand value assignment trend. Moreover, it should understand what type of operations causes generation of values for assignment. That way, analyzer can deduct which values are actual limits for variables. For example, if a variable holds values of a multiplication, it can grow exponentially and even slightest trend for increase in its values can be proved to be a threat.

Deployment is the least worked on phase of waterfall methodology. We strongly believe that there is a good opportunity for further advancements in that area. We especially suggest methods for hot patching binaries in the memory. This way, it will be possible to apply patches without restarting processes or the system. Biggest advantage will be the ability to install latest security patches without any delays (for instance waiting for server maintenance schedules etc).

We also would like to see advancements in tools that automatically track changes that are performed by the applications during their lifetimes. This information can then be used to uninstall applications in a way that they were never existed. The biggest challenge we see in development of such tools is its performance and transparency. Accuracy is another challenge, but can be resolved with cooperation of applications.

As research in regression section shows, industry needs a way to reduce number of regressions caused from vulnerabilities. Increased size of code base and number of supported platforms makes this need high priority. Regressions can be decreased in several ways. One of them is improving maintenance processes. This is generally more costly approach, because it involves new hires or organizational changes. On the other hand, designing software to be maintenance aware could be very cost efficient. Although we are not aware of such an academic paper, some companies (especially Microsoft in order to support its aging operating system Windows) implement some methods. An academic research could help the industry in a broader perspective.

12. Appendix A: Glossary of Terms

This short glossary of terms is used in the thesis.

Black Hole Attack: In this kind of attack, attacker makes a very low cost request to the peer, who will allocate considerable resources to satisfy the request. However, attacker never goes on with the transaction, yet still makes other requests. Attacked program throws even more resources to the sink and finally crashes. Tcp Syn attacks can be an example for this kind of attack.

Buffer Overrun: Attacker exploits vulnerability by writing its own data to a buffer beyond its limits. This causes to overwrite stack of the thread and thus modify it in the way the attacker wants to. Since local variables and return address are kept in the stack, attacker can change internal state of function call or change return address to branch to a desired function. This type of attack can be achieved by leveraging other attacks.

Denial of Service (DoS): Attacker manages to prevent the service providers from servicing anymore. Users experience shortage in the provided services up to the point where there is no service at all. This kind of attacks gives economic damage and bad reputation to the attacked people, however less advantage to the attacker. Therefore, attackers use this attack to stage other attacks or just for fun or reputation.

Integer Overrun: Integer variables, like all other variables, are kept in fixed size storage. Since a certain amount of bits is required to hold certain amount of number combinations, variables maximum values are fixed, too. If someone starts with incrementing an integer value, it will finally exhaust possible combinations and then will silently over wrap to zero. This event is called integer overrun. Integer overrun can happen during almost all types of arithmetic calculations. Because of the incorrect results of calculations, program behaves erratically and may cause vulnerabilities. Integer overflows generally are very hard to exploit. What makes them dangerous is that they open the way for potential buffer overrun vulnerabilities.

Local Attack: This attack type can be only performed locally. This means that a user must be logged on. However, type of user may vary. Some attacks require that at least a normal user to be logged on; other can work with just a guest account. Since local attack implies physical access to the machine, this kind of attacks is considered less critical.

Remote Attack: Attacker can exploit the vulnerability even from a remote system. This of course implies a networked situation and therefore affects only the computers that are connected to the network. However, interestingly, some of these attacks can be performed locally from a loop back interface. Attacking a system that is on the other end world remotely is huge opportunity for attackers. Since no physical access is required, this is the type of attacks, which frightens the system administrators.

Remote Code Execution: Attacker manages to execute his or her own code in the remote system, most probably by leveraging a buffer overrun. Injected code can do whatever the user context that the thread is running can do. If the user context belongs to a less privileged account, attacker is very limited with what can be done. On the other hand, if the context belongs to a highly privileged account, attacker can virtually do everything that he or she wants. In theory, this type of attack goes to the level of performing arbitrary operations like an administrator on the local console.

Privacy Compromise: While remote code execution ability is ultimate goal when attacking a system, a privacy compromise may do as well if this is what required after all. In this situation, attacker manages to make remote reveal the information that it should do normally. Disclosed information can range from system uptime, to the level of personal records of people of corporate records.

Privilege Elevation: Attacker manages to increase its privileges beyond what he or she was granted originally. Then, of course, he or she uses these privileges to make actual attack to the system.

Traffic Analysis Attack: Network traffic of the system is analyzed in terms of its nature, behavior and patterns. Then, the system is attacked by modifying, repeating, analyzing or spoofing the network traffic.

Trojan Horse: (As used in computer science) Attacker injects arbitrary code to the otherwise useful and benign application and convinces attacked user to use that program. Since user thinks that the program is no harmful, he or she accepts is into

the premises of his or her user context by running it. While seemingly no harmful, application performs its damage in the background using the rights and privileges of the user, who ran the application.

13. References

[1] **A. Baratloo, N. Singh, T. Tsai,** 2000, "Transparent Run-Time Defense against Stack Smashing Attacks", Bell Labs

[2] **CNN International Inc,** 2000, "Rebuffed Internet extortionist posts stolen credit card data", http://archives.cnn.com/2000/TECH/computing/01/10/credit.card.crack.2/

[3] **CNN International Inc,** 2001, "Cost of 'Code Red' rising", http://archives.cnn.com/2001/TECH/internet/08/08/code.red.II/index.html

[4] **Wikipedia,** "Keyword: MyDoom", http://en.wikipedia.org/wiki/Mydoom

[5] **Global Melissa Virus Information Center,** http://www.f-secure.com/melissa/

[6] **Schneier B,** 1999, "Applied Cryptography", Wiley Press

[7] **www.SysInternals.com Freeware**, http://www.sysinternals.com

[8] **Symantec Security Response, "**W32.SQLExp.Worm", http://securityresponse.symantec.com/avcenter/venc/data/w32.sqlexp.worm.html

[9] **LeBlanc D,** 2004, "Integer Handling with the C++ SafeInt Class, Microsoft Office" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

[10] **Microsoft TechNet Technical Bulletins,** "Patch Available for "Windows Multithreaded SSL ISAPI Filter" Vulnerability", http://www.microsoft.com/technet/security/bulletin/ms99-053.mspx

[11] **Microsoft Windows Application Compatibility Home Page,** http://www.microsoft.com/windows/appcompatibility/appverifier.mspx

[12] **Sommerville I,** 2004, "Software Engineering (7th Edition)", Addison Wesley

[13] **Thompson H,** 2005, "Application Penetration Testing", IEEE Security & Privacy

[14] **Gamma E, Helm R, Johnson R, Vlissides J,** 1995, "Design Patterns", Addison Wesley Professional

[15] **Howard M, LeBlanc D,** 2004, "Writing Secure Code, Second Edition", Microsoft Press

[16] **McConnell S,** 2004, "Code Complete, Second Edition", Microsoft Press

[17] **Solomon D, Russinovich M,** 2004, "Windows Internals, Second Edition", Microsoft Press

[18] **Viega C, Messier M,** 2003, "Secure Programming Cookbook for C and C++", O'Really

[19] **Chinchani R, Iyer A, Jayaraman B, Upadhyaya S,** 2003, "ARCHERR: Runtime Environment Driven Program Safety", University of Buffalo

[20] **Microsoft Corporation,** Microsoft Security Bulletin MS03-026, **"**Buffer Overrun In RPC Interface Could Allow Code Execution (823980)" http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx

[21] **Symantec Inc.,** Security Response, **"**W32.Blaster.Worm" http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html

[22] **Symantec Inc.,** Security Response, W32.Sobig.F@mm http://securityresponse.symantec.com/avcenter/venc/data/w32.sobig.f@mm.html

[23] **CNN International Money,** "SoBig worm not slowing down yet" http://money.cnn.com/2003/08/21/technology/sobig/?cnn=yes

[24] **Schneier B,** 2005, "Two-Factor Authentication: Too Little, Too Late", Communications of the ACM Volume 48, #4

[25] **US-CERT,** "Summary of Security Items from March 23 through March 29, 2005"
http://www.us-cert.gov/cas/bulletins/SB05-089.html

[26] **Mitnick K, Simon W, Wozniak S,** 2002, "The Art of Deception: Controlling the Human Element of Security", Wiley Press

[27] **Mitnick K, Simon W,** 2005, "The Art of Intrusion: The Real Stories Behind the Exploits of Hackers, Intruders & Deceivers", Wiley Press

[28] **MS SQL Server Security**
http://www.sqlsecurity.com/DesktopDefault.aspx

[29] **Shalloway A, Trott J,** 2001, "Design Patterns Explained: A New Perspective on Object-Oriented Design", Addison Wesley

[30] **Necula, G.C., McPeak, S., Weimer, W,** 2002, "CCured, "Type-safe Retrofitting of Legacy Code."", Proceedings of the Principles of Programming Languages

[31] **Wikipedia,** "Meet-in-the-middle attack"
http://en.wikipedia.org/wiki/Meet-in-the-middle_attack

[32] **Merkle R, Hellman N,** 1981, "On the Security of Multiple Encryption", Stanford University

[33] **Microsoft Developer Network, "**AWE Example"
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/awe_example.asp

[34] **Petzold C,** 2001, "Programming Windows", Microsoft Press

[35] **Grimes R,** 1997, "Professional DCOM Programming", Peer Publishing Inc

[36] **Abernethy R, Morin R, Chahin J,** 1999, "COM/DCOM Unleashed (Unleashed Series)", Sams Publishing

[37] **Maguire S,** 1993, "Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs (Microsoft Programming Series)", Microsoft Press

[38] **Meyer S,** 2005, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)", Addison-Wesley Professional

[39] **Meyer S,** 1995, “More Effective C++: 35 New Ways to Improve Your Programs and Designs”, Addison-Wesley Professional

[40] **Raymond E S,** 2003, “The Art of UNIX Programming”, Addison-Wesley Professional

[41] **Microsoft Corporation,** “PREFast” http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx

[42] **Microsoft Developer Network, “**Using StrSafe.h Functions” http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp

[43] **Zammit J,** 1998, **“**How Do We Build Correct Systems”, Department of Computer Information Systems, Faculty of Science, University of Malta
Also available at: http://www.cis.um.edu.mt/~jzam/building.html

[44] **Control Chaos, “**Development: Empirical or Planned?” http://www.controlchaos.com/old-site/debate.htm

[45] **Microsoft Corporation,** Microsoft Security Bulletin MS05-010, “Vulnerability in the License Logging Service Could Allow Code Execution (885834)” http://www.microsoft.com/technet/security/bulletin/MS05-010.mspx

[46] **Swiderski F, Snyder W,** 2004, “Threat Modeling (Microsoft Professional)”, Microsoft Press

[47] **Rumbaugh, J,** 1995, “What Is A Method”, Journal of Object Oriented Programming

[48] **Dijkstra, E W,** 1976, “A Discipline of Programming”, Prentice Hall

[49] **Hoare R,** 1981, “The emperor’s old clothes (1980 Turing Award Lecture)”, Communications of the ACM

[50] **Dyer D,** 2003, “The Top 10 Ways to get screwed by the "C" programming language”
Also available at http://www.andromeda.com/people/ddyer/topten.html

[51] **www.kernel.org,** "ChangeLog-2.6.12.5"
http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.12.5

[52] **[52] B. W. Kernighan**, 1999, "The Practice of Programming", Addison-Wesley

[53] **Linux: "Kernel Backdoor" Attempt**
http://kerneltrap.org/node/1584

[54] **CERT,** "Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability"
http://www.cert.org/advisories/CA-2002-17.html

[55] **Apache Free Software Organization, "**Apache.Org Security Bulletins"
http://httpd.apache.org/info/security_bulletin_20020617.txt

[56] **CERT,** Vulnerability Note VU#481998 "Apache vulnerable to buffer overflow when expanding environment variables"
http://www.kb.cert.org/vuls/id/481998

[57] **Microsoft Corporation,** "Vulnerability in Message Queuing Could Allow Code Execution (892944)"
http://www.microsoft.com/technet/security/Bulletin/MS05-017.mspx

[58] **Howard M, Lipner Steve,** 2003, "Inside Microsoft Security Push", IEEE Security & Privacy, pp 57-61

[59] **Hintz A,** 2002, "Fingerprinting Websites Using Traffic Analysis", The University of Texas at Austin
Also available at: http://www.freehaven.net/anonbib/cache/hintz02.html

[60] **Microsoft Corporation,** "Buffer Overrun In RPCSS Service Could Allow Code Execution (824146)"
http://www.microsoft.com/technet/security/bulletin/MS03-039.mspx

[61] **CheckPoint**, "Case Study: The Real Cost of Worm Outbreaks"
http://www.checkpoint.com/products/home_promo/worm_outbreaks.html

[62] **Microsoft Corporation,** "Vulnerability in the License Logging Service"
http://www.microsoft.com/technet/security/Bulletin/MS05-010.mspx

[63] **Microsoft Corporation,** "Vulnerability in PNG Processing Could Allow Remote Code Execution (890261)" http://www.microsoft.com/technet/security/Bulletin/MS05-009.mspx

[64] **Microsoft Corporation**, "Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987)" http://www.microsoft.com/technet/security/bulletin/ms04-028.mspx

[65] **Microsoft Corporation**, "Vulnerability in Windows Could Allow Information Disclosure(888302)" http://www.microsoft.com/technet/security/bulletin/ms05-007.mspx

[66] **Common Vulnerabilities And Exposures**, "CAN-2003-0984" http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0984

[67] **Microsoft Developer Network,** SecureZeroMemory(), http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/securezeromemory.asp

[68] **Microsoft Developer Network,** Safe String Reference http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mobilesdk5/html/mob5grfsafestringreference.asp

14. Autobiography

Mehmet Barış Saydağ is born in Istanbul, Turkey in 1980. After finishing Cağaloğlu Gymansium, he studied Computer Engineering in Istanbul Technical University between 1998 and 2002. He started his masters of engineering education in 2002 and continued until 2005. During this time, he worked for Turkish Airlines as one of the administrators of its vast network. His technical background, work experience, personal interests and research inspired him to go ahead and make further research in secure software development. He is now working in Microsoft Corporation as software engineer.