**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# Efficient countermeasures for software vulnerabilities due to memory management errors

Promotoren :
Prof. Dr. ir. W. JOOSEN
Prof. Dr. ir. F. PIESSENS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

**Yves YOUNAN**

mei 2008

**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# Efficient countermeasures for software vulnerabilities due to memory management errors

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

**Yves YOUNAN**

U.D.C. 681.3*D46

mei 2008

D/2008/7515/45
ISBN 978-90-5682-936-0

# Abstract

Despite many years of research and large investments by companies, the development of secure software is still a significant problem. This is evidenced by the steady increase in vulnerabilities that are reported year by year. Fast spreading worms like the Code Red worm, which caused an estimated worldwide economic loss of $2.62 billion, will often exploit implementation errors in programs to spread rapidly.

Vulnerabilities that can be exploited by attackers to perform code injection attacks are an important kind of implementation error. The Code Red worm exploited a buffer overflow to be able to run arbitrary code on the vulnerable machine, allowing it to spread by copying itself to the hosts it infected. The widespread use of C-like languages where such vulnerabilities are an important issue has exacerbated the problem.

In this dissertation we examine a number of vulnerabilities in C-like languages that can be exploited by attackers to perform code injection attacks and discuss countermeasures that provide protection against these kinds of attacks. This dissertation consists of three important parts: it starts off by presenting an extensive survey of current vulnerabilities and countermeasures, this is followed by a discussion of two novel countermeasures which aim to better protect against attacks on different vulnerabilities while having only a negligible impact on performance.

The survey provides a comprehensive and structured survey of vulnerabilities and countermeasures for code injection in C-like languages. Various countermeasures make different trade-offs in terms of performance, effectivity, memory cost, compatibility, etc. This makes it hard to evaluate and compare the adequacy of proposed countermeasures in a given context. This survey defines a classification and evaluation framework, on the basis of which advantages and disadvantages of countermeasures can be assessed. Based on the observations and the conclusions that were drawn from the survey, two countermeasures have been designed, implemented and evaluated.

The first countermeasure we present is an efficient countermeasure against stack smashing attacks. Our countermeasure does not rely on secret values (such as canaries) and protects against attacks that are not addressed by state-of-the-art countermeasures. Our technique splits the standard stack into multiple stacks. The allocation of data types to one of the stacks is based on the chances that a specific data element is either a target or source of attacks. We have implemented our solution in a C-compiler for Linux. The evaluation shows that the overhead of using our countermeasure is negligible.

The second countermeasure protects against attacks on heap-based buffer overflows and dangling pointer references. Overwriting the management information of the memory allocation library is often a source of attack on these vulnerabilities. All existing countermeasures with low performance overhead rely on magic values, canaries or other probabilistic values that must remain secret. In the case of magic values, a secret value is placed before a crucial memory location and by monitoring whether the value has changed, overruns can be detected. Hence, if attackers are able to read arbitrary memory locations, they can bypass the countermeasure. This countermeasure presents an approach that, when applied to a memory allocator, will protect against this attack vector without resorting to magic. We implemented our approach by modifying an existing widely-used memory allocator. Benchmarks show that this implementation has a negligible, sometimes even beneficial, impact on performance.

*To my mother, Yolande De Moor.*

# Acknowledgements

This dissertation marks the conclusion of several years of research at the DistriNet research group of Department of Computer Science of the Katholieke Universiteit Leuven. I have met many interesting people willing to share their knowledge or offer help. My gratitude goes out to all of them. First and foremost I would like to thank my advisors Professor Wouter Joosen and Professor Frank Piessens for giving me the chance to pursue a PhD. Their valuable insight and advice has helped improve the quality of the work that you will find in this dissertation. My thanks also go out to Professor Pierre Verbaeten and Professor Bart Demoen for serving in my PhD-advisory committee during the last four years. I would also like to thank Professor R. Sekar and Professor Ùlfar Erlingsson for agreeing to serve as members of my jury and Professor Dirk Vandermeulen for chairing the jury. I have met many people at the Department of Computer Science which have made working there an interesting and fun experience. I'd like to thank everyone of my current colleagues and former colleagues. At the risk of missing someone, I will try and list those people who have made a difference: Koen Buyens, Maarten Bynens, Thomas Delaet, Liesje Demuynck, Kris Demarsin, Dr. Lieven Desmet, Dr. Bart De Win, Bart Elen, Kristof Geebelen, Tom Goovaerts, Johan Gregoire, Thomas Heyman, Wouter Horré, Aram Hovsepyan, Dr. Bart Jacobs, Dr. Nico Janssens, Eryk Kulikowski, Bert Lagaisse, Dr. Tom Mahieu, Dr. Sam Michiels, Adriaan Moors, Dr. Julien Pauty, Pieter Philippaerts, Davy Preuveneers, Dr. Peter Rigole, Dr. Riccardo Scandariato, Jan Smans, Tom Stijnen, Dr. Eddy Truyen, Dr. Yves Vandewoude, Dr. Marko Vandooren, Dries Vanoverberghe, Dr. Tine Verhanneman, Dr. Kris Verlaenen, Kristof Verslype, Koen Victor, Frédéric Vogels, Dr. Andrew Wills, Kim Wuyts and Koen Yskout. A special thank you also goes out to all those of you who participated in the capture flag sessions.

During the course of the work which culminated in this PhD, I ended up spending 6 months at the Secure Systems Lab at Stony Brook University. I would like to thank Professor R. Sekar in particular for his advice during my stay. I would also like to thank the people who made my stay an enjoyable and interesting one: Ezio Bartocci, Dr. Lorenzo Cavallaro, Munyaradzi Chiwara, Agata Cwalina, Carmelo Fruciano, Cheryl Lassman, Srivanni Narra, Oliviero Riganelli, Indira Rocha, Weiqing Sun, Alok Tongaonkar and Jun (Jamie) Yuan. I have also had the pleasure of collaborating with Dr. Davide Pozza during his 3 month research visit in Leuven.

I am also grateful to the people from nologin for their help and for the interesting technical discussions we have had over the years: Martin Zeiser, Craig Williams, Ilja Van Sprundel, Chad Thurnberg, Stephanie Smith, Shyama Rose, Orlando Padilla, H D Moore, Matt Miller, Jan Muenther, Richard Johnson, Ken Johnson, David Hulton, Daniel Hodson, Jordan Hind, Lurene Grenier, Blake Frantz, Jon Elch, Jerry Connolly, Adam Cecchetti, David (Brian) Cavenah, Chris Carr, Eric Cabetas and Neil Archibald.

And last but not least, I would like to thank my friends and family for their support, especially my girlfriend Miet.

Yves Younan
May 2008

# Contents

**Contents** **vii**

**List of Figures** **xi**

**List of Tables** **xiii**

**List of Listings** **xv**

**1 Introduction** **1**

1.1 Scope of the dissertation . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 Main contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.2.1 Survey of vulnerabilities and countermeasures for C and C++ 7
1.2.2 A countermeasure against attacks on stack-based buffer overflows . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.2.3 A countermeasure against attacks on heap-based buffer overflows . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.2.4 Other contributions . . . . . . . . . . . . . . . . . . . . . . 8
1.3 Overview of the dissertation . . . . . . . . . . . . . . . . . . . . . . 8

**2 Code injection in C and C++ : a survey of vulnerabilities and countermeasures** **11**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.2 Implementation vulnerabilities and exploitation techniques . . . . . 15
2.2.1 Buffer overflows . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2.2 Dangling pointer references . . . . . . . . . . . . . . . . . . 23
2.2.3 Format string vulnerabilities . . . . . . . . . . . . . . . . . 27
2.2.4 Integer errors . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.3 Countermeasure properties . . . . . . . . . . . . . . . . . . . . . . . 29
2.3.1 Type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.3.2 Vulnerabilities . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.3.3 Protection level . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.3.4 Usability . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.3.5 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.3.6 Computational and memory cost . . . . . . . . . . . . . . . 34
2.4 Countermeasures . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.4.1 Safe languages . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.4.2 Bounds checkers . . . . . . . . . . . . . . . . . . . . . . . . 38
2.4.3 Probabilistic countermeasures . . . . . . . . . . . . . . . . . 40
2.4.4 Separation and replication of information . . . . . . . . . . 44
2.4.5 Paging-based countermeasures . . . . . . . . . . . . . . . . 44
2.4.6 Execution monitors . . . . . . . . . . . . . . . . . . . . . . . 46
2.4.7 Hardened libraries . . . . . . . . . . . . . . . . . . . . . . . 49
2.4.8 Runtime taint trackers . . . . . . . . . . . . . . . . . . . . . 50
2.4.9 Dynamic analysis and Testing . . . . . . . . . . . . . . . . . 51
2.4.10 Static Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 51
2.5 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
2.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**3 Extended protection against stack smashing attacks without performance loss 57**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
3.2 Stack-based buffer overflows . . . . . . . . . . . . . . . . . . . . 60
3.3 The multiple stacks countermeasure to protect against buffer overflow vulnerabilities . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.3.1 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.3.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 68
3.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
3.4.1 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 70
3.4.2 Memory overhead . . . . . . . . . . . . . . . . . . . . . . . 72
3.5 Discussion and ongoing work . . . . . . . . . . . . . . . . . . . . 73
3.6 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
3.6.1 Protection from attacks on stack-based vulnerabilities . . . 74
3.6.2 Alternative approaches . . . . . . . . . . . . . . . . . . . . . 75
3.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

**4 Improving memory management security for C and C++ 79**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.2 Heap-based vulnerabilities for code injection attacks . . . . . . . . 82
4.2.1 Heap-based buffer overflow . . . . . . . . . . . . . . . . . . 82
4.2.2 Off by one errors . . . . . . . . . . . . . . . . . . . . . . . . 83
4.2.3 Dangling pointer references . . . . . . . . . . . . . . . . . . 83
4.3 Memory managers . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.3.1 Doug Lea's memory allocator . . . . . . . . . . . . . . . . . 84
4.3.2 Boehm garbage collector . . . . . . . . . . . . . . . . . . . . 90

4.3.3 Summary . . . 92
4.4 A more secure memory allocator . . . 92
4.4.1 Countermeasure Design . . . 93
4.4.2 Prototype Implementation . . . 96
4.4.3 Managing chunk information . . . 98
4.5 Evaluation . . . 98
4.5.1 Performance . . . 100
4.5.2 Memory overhead . . . 102
4.5.3 Security evaluation . . . 104
4.6 Related work . . . 105
4.6.1 Protection from attacks on heap-based vulnerabilities . . . 105
4.6.2 Alternative approaches . . . 108
4.7 Conclusion . . . 110

**5 Conclusion** **111**
5.1 Contributions . . . 111
5.2 Future work . . . 113
5.2.1 A bounds checker for pointer arithmetic . . . 113
5.2.2 A countermeasure for dangling pointer references . . . 113
5.3 Future research opportunities and application domains . . . 114
5.3.1 Embedded systems and mobile devices . . . 114
5.3.2 Virtual machine monitors . . . 115

**Bibliography** **116**

**List of Publications** **137**

**Biography** **139**

**Dutch Summary** **141**

# List of Figures

1.1 Top 20 Threats and Vulnerabilities, January through October 2007 (source: [85], ©Cisco Systems Inc.) . . . 3
1.2 Shifts in Threats and Vulnerabilities Reported (source: [85], ©Cisco Systems Inc.) . . . 4
1.3 Countermeasure triangle . . . 5

2.1 Stack-layout on the IA32-architecture . . . 17
2.2 Normal stack-based buffer overflow . . . 18
2.3 Stack-based buffer overflow overwriting frame pointer . . . 18
2.4 Stack-based buffer overflow using indirect pointer overwriting . . . 19
2.5 Heap containing used and free chunks . . . 21
2.6 Heap-based buffer overflow . . . 22
2.7 List of free chunks . . . 24
2.8 *Chunk*4 added to the list of free chunks (*chunk*3 not shown) . . . 25
2.9 List of free chunks with chunk2 freed twice . . . 26
2.10 *Chunk*2 reallocated as used chunk . . . 26
2.11 Overwriting the return address using a double free . . . 27
2.12 Overwriting a value 1 byte at a time using %n specifiers . . . 28

3.1 Indirect pointer overwriting attack . . . 62
3.2 Stack layout for 5 stacks . . . 68
3.3 Gaps on the different stacks . . . 69

4.1 Heap containing used and free chunks . . . 85
4.2 Heap-based buffer overflow in *dlmalloc* . . . 86
4.3 Off by one error in *dlmalloc* . . . 88
4.4 List of free chunks in dlmalloc: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred. 88
4.5 Linked list of free chunks in Boehm's garbage collector . . . 92
4.6 Double free of *chunk2* in Boehm's garbage collector . . . 92
4.7 Original (left) and modified (right) process memory layout . . . 94

4.8 *Lookup table* and chunkinfo layout . . . . . . . . . . . . . . . . . . . 97

# List of Tables

2.1 Countermeasure type . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.2 Vulnerability addressed by a countermeasure . . . . . . . . . . . . 31
2.3 Protection level of a countermeasure . . . . . . . . . . . . . . . . . 31
2.4 Stage of the software engineering process that a countermeasure is applied at . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.5 Effort required to apply a countermeasure . . . . . . . . . . . . . . 33
2.6 Limitations of a countermeasure . . . . . . . . . . . . . . . . . . . 34
2.7 Computational and memory cost of a countermeasure . . . . . . . 35
2.8 Safe languages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.9 Bounds Checkers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.10 Probabilistic countermeasures . . . . . . . . . . . . . . . . . . . . . 44
2.11 Separation Countermeasures . . . . . . . . . . . . . . . . . . . . . . 45
2.12 Paging-based countermeasures . . . . . . . . . . . . . . . . . . . . 47
2.13 Execution monitors . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
2.14 Hardened Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
2.15 Runtime taint trackers . . . . . . . . . . . . . . . . . . . . . . . . . 50
2.16 Dynamic Analysis and Testing . . . . . . . . . . . . . . . . . . . . 52
2.17 Static Analyzers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

3.1 Attack source versus attack target categories . . . . . . . . . . . . 64
3.2 Benchmark results of the multistack approach . . . . . . . . . . . . 71
3.3 Local variables stored on each stack . . . . . . . . . . . . . . . . . 71

4.1 Programs used in the evaluations of *dnmalloc* . . . . . . . . . . . . 99
4.2 Average macrobenchmark runtime results for *dlmalloc* and *dnmalloc* 100
4.3 Average microbenchmark runtime results for *dlmalloc* and *dnmalloc* 101
4.4 Average memory usage for *dlmalloc* and *dnmalloc* . . . . . . . . . 103
4.5 Results of exploits against vulnerable programs protected with *dnmalloc* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

# List of Listings

2.1 Unlink macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 Unlink macro expanded . . . . . . . . . . . . . . . . . . . . . . . . 23
2.3 Adding a chunk to the list of free chunks . . . . . . . . . . . . . . . 24
2.4 Freeing chunk2 twice . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.5 Unlinking chunk2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.1 Unlink macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.2 Unlink macro expanded . . . . . . . . . . . . . . . . . . . . . . . . 87
4.3 Adding a chunk to the list of free chunks . . . . . . . . . . . . . . . 89
4.4 Freeing chunk2 twice . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.5 Unlinking chunk2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.6 Example microbenchmark (no memset) . . . . . . . . . . . . . . . . 101

# Chapter 1

# Introduction

Since the advent of multi-user computing, security has been an important concern. In the early days of multi-user computing, users would attempt to get extra time on, or access to, shared resources. This spawned a whole domain of research into computer security that has become an integral part of the computer science research field. The advent of massively networked systems like the Internet gave the need for security a renewed urgency. Researchers have been quite successful in designing security mechanisms that will provide essential protection to computer systems. The domain of access control, for example, has built extensive models for providing access control depending on the specific access control needs of an entity. Research into cryptography has also produced extensive and provably secure cryptographic algorithms that will keep important data unreadable as long as the key is kept safe.

While such important security problems have been addressed and are still being improved upon, a very important problem remains with the implementation of programs. Often implementation errors will undo the security provided by access control and cryptographic protocols. In many cases, errors in implementing a cryptographic protocol will significantly weaken the algorithm or errors in the implementation of access control mechanisms could allow users to elevate their privileges.

An important form of implementation errors that leads to security problems also results from a lack of input checking. This lack of input checking can result in data being interpreted as computer code. This can occur both for interpreted languages as for compiled languages. However, such vulnerabilities have been particularly severe for programs that were written in C-like languages, because these programs are often used for network daemons, operating systems and device drivers. In C-like languages, insufficient input checking could lead to a buffer overflow, where a vulnerable program will write past the end of an object, overwriting adjacent objects.

This situation has improved with the advent of safe languages, such as Java and C#, which give programmers less direct access to memory and can thus prevent specific bugs from occurring. Garbage collecting languages, do not allow programmers to manually free memory, removing the problem of dangling pointer references. Safe languages will usually not allow the programmer to directly manipulate pointers or perform pointer arithmetic, preventing an important cause of buffer overflows. Array accesses via indexes are often checked at runtime by storing the size of the array and ensuring that the index access is within the bounds of the array.

However, it is not always possible to use such safe languages. There is a significant amount of legacy code that has been written in C-like languages that is still in use today. Moreover, many programmers have expertise in these languages and continue to use them to develop new products in these languages. In some cases, the use of a C-like language is a necessity: for specific system software, programmers need direct access to memory, which would be denied by safe languages. Some devices also have very specific constraints with respect to memory usage and performance, again making a C-like language an attractive choice.

Vulnerabilities that are the result of memory management errors in C-like languages, are a significant threat to the security of present day computer systems. Most of these vulnerabilities result from mishandling of arrays, resulting in buffer overflow vulnerabilities. Using such a vulnerability, an attacker can overwrite memory locations that the execution environment relies on to correctly execute programs.

This problem occurs because abstractions that exist in a higher-level language do not exist in a lower-level representation of that program [60]. When a C program is compiled, the compiler will introduce a number of mechanisms that allow it to more easily facilitate program execution and implement these high level abstractions. These mechanisms are not present in the C language and the programmer should not directly access them. However, because C is an unsafe language, it is possible for a program to access these mechanisms, either directly by pointer manipulation or accidentally by a vulnerability. When such a vulnerability occurs, an attacker may be able to use it to gain control of the program's execution flow. An example of such a mechanism is the return address: this address is used to be able to execute functions. When a function is called, the address of the next instruction after the function call is placed onto the stack. Once the function has finished executing, execution of the main program will resume at the return address. This mechanism allows the program to execute nested and recursive function calls. An attacker can abuse this mechanism using a vulnerability to modify the return address to point to a different location. When the function terminates, it will transfer control to this new location and any data stored there will be interpreted as machine code and executed. As such, if attackers are able to provide input to the program that will be stored in the program's memory and

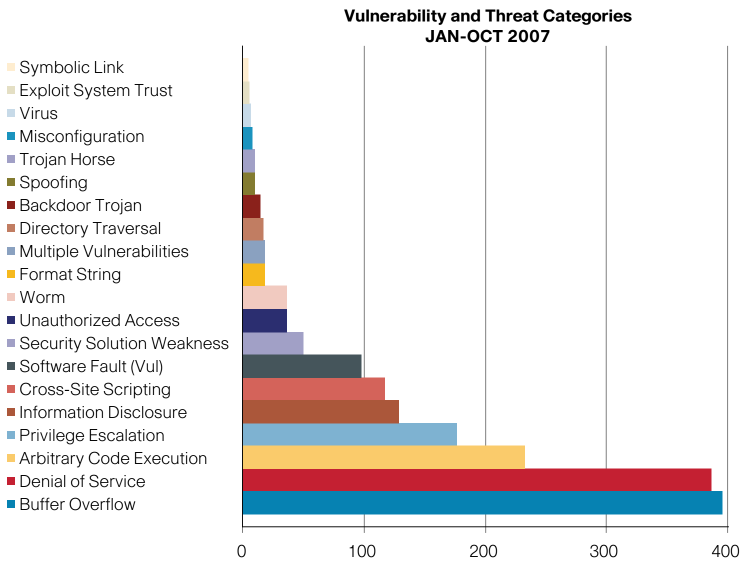


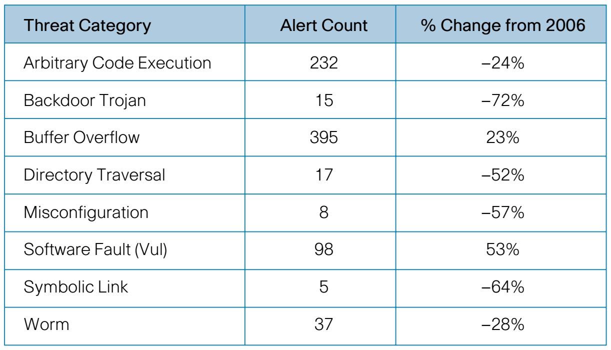
Figure 1.1: Top 20 Threats and Vulnerabilities, January through October 2007 (source: [85], ©Cisco Systems Inc.)

are then able to use a vulnerability to modify the return address, they will be able to execute arbitrary code with the privilege level of the process.

This type of vulnerability was exploited in many of the most devastating worms in recent memory: the Code Red worm, which caused a world wide economic loss estimated at $ 2.62 billion [58]; the Sasser worm, which caused Delta Airlines to cancel several transatlantic flights and shut down X-ray machines at a Swedish hospital [185]; and the Zotob worm, which most likely caused a nationwide breakdown of the computers handling the DHS's US-VISIT program on August, 18th 2005, leading to long lines of travelers at several major airports [176].

According to the NIST's National Vulnerability Database [120], 584 buffer overflow vulnerabilities were reported in 2005, making up 12% of the 4852 vulnerabilities reported that year. In 2004 the amount of reported buffer overflow vulnerabilities was 341 (14% of 2352). This means that, while the amount of reported vulnerabilities almost doubled in 2005, buffer overflows still remain an important source of attack. 418 of the 584 buffer overflows reported in 2005 had a high severity rating, this makes up 21% of the 1923 vulnerabilities rated with a high severity level. They also make up 42% of the vulnerabilities that allow an attacker to gain administrator access to a system.

In Cisco's annual report for 2007 [85], statistics about vulnerabilities and their



| Threat Category | Alert Count | % Change from 2006 |
|---|---|---|
| Arbitrary Code Execution | 232 | –24% |
| Backdoor Trojan | 15 | –72% |
| Buffer Overflow | 395 | 23% |
| Directory Traversal | 17 | –52% |
| Misconfiguration | 8 | –57% |
| Software Fault (Vul) | 98 | 53% |
| Symbolic Link | 5 | –64% |
| Worm | 37 | –28% |

Figure 1.2: Shifts in Threats and Vulnerabilities Reported (source: [85], ©Cisco Systems Inc.)

causes are presented based on the alerts issued by the Cisco Security IntelliShield Alert Manager Service[1]. In the graph in Figure 1.1 the vulnerabilities and their effects are mixed together, making it harder to gather useful data. However, it shows clearly that vulnerabilities that could allow arbitrary code execution when exploited make up the second largest group of vulnerabilities reported by the IntelliShield Alert Service in 2007. It also gives a clear indication that buffer overflow vulnerabilities are still a very important vulnerability, making up the largest group of vulnerabilities reported by the IntelliShield Alert Service.

In the report, a comparison is also made to the trends with vulnerabilities in 2006 (see Figure 1.2). While less vulnerabilities were reported that resulted in code execution, more vulnerabilities were reported that are buffer overflows. This could mean that more countermeasures are being applied, which turn buffer overflows into denial of service attacks, rather than into code injection attacks. While this is a positive trend demonstrating the positive results of the research in this area, it is also worrying that the relative amount of buffer overflows has increased and makes up such a large share of the reported vulnerabilities.

## 1.1 Scope of the dissertation

In this dissertation we focus on some important vulnerabilities that can be used by an attacker to gain code execution in programs written in C-like languages:

**Buffer overflows** can occur when a program does not ensure that a write op-

[1]This is a service that will gather information on current vulnerabilities and distribute it to customers who subscribe to the service.

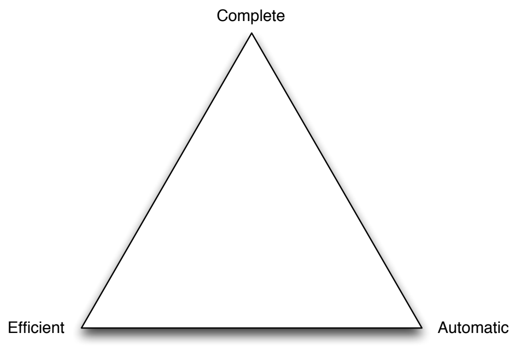


Figure 1.3: Countermeasure triangle

eration will not write past end of the memory that was allocated for an object

**Dangling pointer references** can occur when pointers exist to memory that has been released back to the system.

**Format string vulnerabilities** can occur when an attacker has control over a format string that is later passed to a format function.

**Integer errors** can occur when integers overflow or a signed integer is interpreted as an unsigned integer.

We will often refer to the exploitation of these vulnerabilities as code injection attacks. This is the term we chose to describe these kinds of attacks at the start of the research. It has since been redefined by the security community as pertaining mostly to web application vulnerabilities where interpreted code is injected into a web application [184], while the vulnerabilities described in this dissertation are now often referred to as memory corruption vulnerabilities. In this dissertation, the term code injection attack refers to the exploitation of the aforementioned vulnerabilities.

Many countermeasures have been designed to address these vulnerabilities. These countermeasures are usually evaluated using three important criteria: completeness, efficiency and the level of automation. In analogy with the project triangle in design engineering "Good, Cheap, Reliable: Pick any two" [183], most countermeasures will only satisfy two of the criteria.

**Completeness** defines how complete the protection is against the vulnerability that the countermeasure is addressing. For example, bounds checkers will generally be complete with respect to buffer overflows, because they will prevent access to memory out of the bounds of the object that is being addressed.

**Efficiency** determines the countermeasure's impact on both performance and memory usage.

**Automatic** covers multiple facets of countermeasures that could require human intervention. The less manual intervention is required, the more automatic the countermeasure will be. If a countermeasure is not compatible with existing C code, it will require manual intervention. Countermeasures that require interpretation and processing of the results (like static and dynamic analyzers), also require some intervention. If a program must be annotated for the countermeasure to work, it will also need manual intervention.

Each countermeasure is a compromise between efficiency, completeness and automatization. Often two of these properties can be satisfied well, but compromises have to be made in the other field:

- A countermeasure that is efficient and complete will require manual intervention to be applied to a program.
- A countermeasure that is complete and can be applied automatically, will be inefficient.
- A countermeasure that is efficient and can be applied automatically, will be incomplete.

However, this is the ideal situation when discussing countermeasures: sometimes it is possible to provide some improvement for countermeasures in one area while only minimal or no loss is made in the others. The countermeasures presented in this dissertation are of that nature: they will provide significant improvements to the completeness of a countermeasure while only slightly (or not at all) decreasing efficiency.

## 1.2 Main contributions

The focus for this dissertation is on designing countermeasures that improve completeness of countermeasures that are very efficient and completely automatic. The research for this dissertation started by performing an extensive survey of vulnerabilities and countermeasures for C and C++, where we defined a classification and evaluation framework for countermeasures. Based on this survey, two countermeasures were designed, one that protects against attacks on stack-based buffer overflows and one that protects against attacks on heap-based buffer overflows.

Both countermeasures are based on a similar premise: separation of control-flow data from regular data. Both countermeasures offer better protection than countermeasures with comparable efficiency and that don't require manual intervention. The focus for this thesis is on countermeasures for buffer overflows, although the heap-based buffer overflow countermeasure also protects against exploitation of some dangling pointer references. The main reason for focusing on buffer overflows is because these remain the most important of the four vulnerabilities that this dissertation addresses [85]. Format string vulnerabilities were an important vulnerability when attackers first discovered how these could be exploited, however the fixes for these kind of vulnerabilities are relatively easy to implement, which is also evidenced by the sharp reduction in reported vulnerabilities after the initial discovery. Integer errors are not exploitable errors by themselves but can lead to buffer overflows when such an integer is used as an offset to a pointer or array.

### 1.2.1 Survey of vulnerabilities and countermeasures for C and C++

A comprehensive and structured survey of vulnerabilities and countermeasures for code injection in C and C++ was performed. Various countermeasures make different trade-offs in terms of performance, effectiveness, memory cost, compatibility, etc. This makes it hard to evaluate and compare the adequacy of proposed countermeasures in a given context. This survey defines a classification and evaluation framework, on the basis of which advantages and disadvantages of countermeasures can be assessed.

### 1.2.2 A countermeasure against attacks on stack-based buffer overflows

An important contribution is a countermeasure for stack-based buffer overflows. In this countermeasure, control-flow data (data that is used to regulate the control flow of the program, like a return address) is separated from regular data on the stack. This separation is achieved by assigning two values to each type of data: target value (how valuable is this type of data to an attacker when trying to perform a code injection attack) and source value (how likely is an attacker to use this type of data to perform an attack). For example, the return address (and other saved registers) has a high target value, since an attacker who controls it can use it to perform code injection. The return address also has a low source value; attackers will never have direct control over it, an attack is needed to modify it. An array of characters has a low target value since attackers can generally not achieve code injection when overwriting such an array. It does, however, have a high source value: these types of arrays are generally the ones vulnerable to buffer overflows. Depending on these types of data the stack can be divided into multiple stacks that are separated from one another. As a result, buffer overflows in an array of characters can only overwrite other arrays of characters, but not return addresses. This countermeasure is very efficient, automatic and more complete than countermeasures that focus on these first two properties.

### 1.2.3 A countermeasure against attacks on heap-based buffer overflows

Another important countermeasure that has been designed and implemented makes it harder for an attacker to execute a code injection attack if a heap-based buffer overflow exists. When a heap-based buffer overflow is exploited, an attacker will often modify the memory management information to reliably achieve code injection. The countermeasure prevents this type of attack by separating the memory management information from the rest of the dynamically allocated memory. The countermeasure is also very efficient, automatic and more complete than other countermeasures that focus on these first two properties.

### 1.2.4 Other contributions

A number of other contributions have also been made during the research performed that resulted in this dissertation:

- A more structured approach to designing countermeasures for code injection attacks is presented in [199].
- A countermeasure for data- and bss overflows is discussed in [202]. This countermeasure has not been implemented in a prototype because the technique that could be applied here is similar to the techniques used for our stack-based countermeasure.
- A discussion of how an attacker could exploit a number of different memory allocators is presented in [203].

## 1.3 Overview of the dissertation

This dissertation is a combination of a number of papers that were published during the last 4 years.

- Chapter 2 contains a survey of the domain of vulnerabilities that can result in code execution and countermeasures that try to prevent, detect or mitigate attacks on these vulnerabilities. It was submitted to ACM Computing Surveys for review and currently remains as such.

- Chapter 3 presents a countermeasure for stack-based buffer overflows that was published in the proceedings of the Twenty-Second Annual Computer Security Applications Conference.
- Chapter 4 presents a countermeasure that was designed and implemented during the research that led to this dissertation. It discusses a technique that can be applied to memory allocators to prevent heap-based buffer overflows. The paper presented in this chapter is a revised version of a paper that was published in the proceedings of the Eighth International Conference on Information and Communication Security. This revised version is currently under submission to the Journal of Computer Security.
- Chapter 5 summarizes the main contributions of the dissertation. It also discusses possible avenues for future work and examines future research directions.

Chapter 2

# Code injection in C and C++ : a survey of vulnerabilities and countermeasures

*In this chapter, we present a survey of vulnerabilities and countermeasures for C and C++. Such vulnerabilities are often used to inject code into a vulnerable program, allowing attackers to execute arbitrary code with the vulnerable program's privileges. The survey extensively discusses how attackers exploit these vulnerabilities and examines the current state of the art in the field of countermeasures for these vulnerabilities. It defines a number of criteria that can be used to classify and evaluate countermeasures for code injection attacks. A significant number of countermeasures were studied to present this survey, the authors aimed to achieve an as complete view as possible of the domain of vulnerabilities that could lead to code injection and their countermeasures.*

*The survey grew out of a study of the related work performed by the author when commencing the research for this dissertation. Based on this survey, a methodology was envisioned that could be used to more effectively design countermeasures, which was published in [199]. This methodology, together with the survey, also gave rise to the idea of separating control data from regular data, which was then used as the basis for designing the two countermeasures that are discussed in chapters 3 and 4.*

*This paper was written together with Wouter Joosen and Frank Piessens. A preliminary version of this paper was published as Technical report CW386 of the Department of Computer Science of the Katholieke Universiteit Leuven in July 2004 [198]. A revised version was submitted to ACM Computing Surveys for review in January 2005. It was conditionally accepted provided minor revisions were performed in March 2007. The revised paper was submitted in May 2007. It currently remains under review. During this revision the countermeasures that are described in this dissertation where also discussed in the survey: both fall into the same category of separation countermeasures but address different types of vulnerabilities.*

## Abstract

The lack of memory-safety in C or C++ often leads to security vulnerabilities. Often, such vulnerabilities are exploited to gain control over the execution-flow of applications via code injection attacks. This paper provides a comprehensive and structured survey of vulnerabilities and countermeasures for code injection in these languages. Various countermeasures make different trade-offs in terms of performance, effectivity, memory cost, compatibility, etc. This makes it hard to evaluate and compare the adequacy of proposed countermeasures in a given context. This paper defines a classification and evaluation framework, on the basis of which advantages and disadvantages of countermeasures can be assessed.

## 2.1 Introduction

Software vulnerabilities are currently, and have been since the advent of multiuser and networked computing, a major cause of computer security incidents [117, 164, 206]. Most of these software vulnerabilities can be traced back to a few mistakes that programmers make over and over again [131]. Even though many documents and books [83, 175, 182] exist that attempt to teach programmers how to program more securely, the problem persists and will most likely continue to be a major problem in the foreseeable future [160]. This document focuses on a specific subclass of software vulnerabilities: implementation errors in C [95] and C++ [59, 167] as well as the countermeasures that have been proposed and developed to deal with these vulnerabilities. More specifically, implementation errors that allow an attacker to break memory safety and execute foreign code are addressed in this report.

Several preventive and defensive countermeasures have been proposed to combat exploitation of common implementation errors and this document examines many of these. Our main goal has been to provide a complete survey of all existing countermeasures. However, since this is an active field of research, our survey is only a snapshot of what has been released up to now. We also describe several ways in which some of the proposed countermeasures can be circumvented. Although some countermeasures examined here protect against the more general case of buffer overflows, this document focuses on examining protection against attacks where an attacker specifically attempts to execute code that an application would not execute in normal circumstances (e.g., injecting code, calling a library function with specific arguments).

Given the large number of countermeasures that have been proposed to deal with code injection attacks, and given the wide variety in techniques used in the design of these countermeasures, it is hard for an outsider of the research field itself to get a good understanding of existing solutions. This paper aims to provide such an understanding to software engineers and computer scientists without specific security expertise, by providing a structured classification and evaluation framework. At the top level, we classify existing countermeasures based on the main technique they use to address the problem. There is some overlap in these sections however and sometimes it may be possible to classify a countermeasure in one category or another.

**Safe languages** are languages in which most of the discussed implementation vulnerabilities have been made hard or impossible. These languages generally require a programmer to specifically implement a program in this language or to port an existing program to this language. We will focus on languages that are similar to C i.e., languages that stay as close to C and C++ as possible: these are mostly referred to as safe dialects of C. Programs written in these dialects generally have some restrictions in terms of memory management: the programmer no longer has explicit control over the dynamic memory allocator.

**Bounds checkers** perform bounds checks on array and pointer operations and detect when the program tries to perform an out of bounds operation and take action accordingly.

**Probabilistic countermeasures** make use of randomness to make exploitation of vulnerabilities harder.

**Separation and replication of information** countermeasures replicate valuable control-flow information or separate control-flow data from regular data. The replicated information can later be used to verify that the original value is unchanged. Separation can prevent regular data from overwriting control-flow data, since these types o‘f data will no longer be adjacent.

**Runtime monitors** monitor specific security relevant events (like system calls) and perform specific actions based on what is monitored. Some monitors will try to limit the damage a successful attack on a vulnerability could do to the underlying system by limiting the actions a program can perform. Others will detect if a program is exhibiting unexpected behavior and will provide alerts if this occurs. The first type of runtime monitor is called a sandbox, while the second type of monitoring is called anomaly detection.

**Hardened libraries** replace library functions with versions that perform extra checks to ensure that the values are correct.

**Runtime taint trackers** will instrument the program to mark input as tainted. If such tainted data is later used in the program where untainted data is expected or is used to modify a trusted memory location (like a return address), then a fault is generated.

**Dynamic analysis and testing tools** instrument the program to generate specific events when a possible error is encountered or will try and cause errors in the program by giving the program various ranges of input.

**Static source code analyzers** attempt to find implementation vulnerabilities by analyzing the source code of an application. This could be as simple as looking for library functions known to be vulnerable to an implementation error or as complicated as making a full model of the program and then deciding what constructs might cause a specific vulnerability.

This document is structured as follows: Section 2.2 contains an overview of the implementation errors that the countermeasures in Section 2.4 attempt to defend against. It also describes typical ways in which these implementation errors can be abused. Section 2.3 contains a description of the properties that we will assign to the various countermeasures that are examined in Section 2.4. Section 2.4 contains our survey of the countermeasures for the vulnerabilities in Section 2.2 and in some cases, ways in which they can be circumvented Section 2.5 examines related work in the field of vulnerability and countermeasure surveys. Section 2.6 presents our conclusion.

## 2.2 Implementation vulnerabilities and exploitation techniques

This section contains a short summary of the implementation errors for which we shall examine countermeasures, it is structured as follows: for every vulnerability we first describe why a particular implementation error is a vulnerability. We then describe the basic technique an attacker would use to exploit this vulnerability and then discuss more advanced techniques if appropriate. We mention the more advanced techniques because some of these can be used to circumvent some countermeasures. A more thorough technical examination of the vulnerabilities and exploitation techniques (as well as a technical examination of some countermeasures) can be found in [195].

When we describe the exploitation techniques in this section we focus mostly on the IA32-architecture [86]. While the details for exploiting a specific vulnerabilities are architecture dependent, the main techniques presented here should be applicable to other architectures as well.

### 2.2.1 Buffer overflows

When an array is declared in C or C++, space is reserved for it and the array is manipulated by means of a pointer to the first byte. At run-time no information about the array size is available and most C-compilers will generate code that allows a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in such adjacent memory space, it could be possible for an attacker to overwrite it.

We have divided this section into two specific types of overflows depending on where in memory they occur. While they are variants of the same vulnerability, we have divided them into two subgroups because they way in which they are exploited and the way in which some countermeasures offer protection against these exploits, can be very different.

### Stack-based buffer overflows

**Vulnerability** On the IA32-architecture the stack grows down in memory (meaning newer stackframes and variables are at lower addresses than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments to a function that was called, registers whose values must be stored across function calls, local variables, the saved frame pointer[1] and the return address. An array allocated on the stack will usually be contained in the section of local variables of a stackframe. If a program copies past the end of this array it will be able to overwrite anything stored before it and it will be able to overwrite the function's management information, like the return address.

Figure 2.1 shows an example of a program's stack when executing the function $f1$. This function was called by the function $f0$ that has placed the arguments for $f1$ after its local variables and then executed a call instruction. The call has saved the return address (a pointer to the next instruction after the call to $f1$) on the stack. The function prologue (a piece of code that is executed before a function is executed) then saved the old frame pointer on the stack. The value of the stack pointer at that moment has been saved in the frame pointer register. Finally, space for two local variables has been allocated: a pointer pointing to data and an array of characters (a buffer). The function would then execute as normal. The colored part of Figure 2.1 indicates what could be written to by the function if the buffer is used correctly.

In Windows, the default exception handler is also stored at the end of the stack. This exception handler contains function pointers to functions that must be executed when a specific exception occurs. An attacker could also overwrite the exception handler by overflowing past the end of an array. Attackers can use this to create a more stable exploit, if the return address is wrong, an exception

[1]As variables get pushed and popped from the stack, the stack pointer keeps changing. If the compiler wants to access a local variable (as an offset from the stack pointer), it must keep track of the current value of stack pointer to be able to access the variable and calculate the offset accordingly. To prevent this calculation the frame pointer is used. When a new stack frame is created, that location is stored in a register. When the compiler tries to access a local variable, it can now use a fixed offset from the frame pointer.

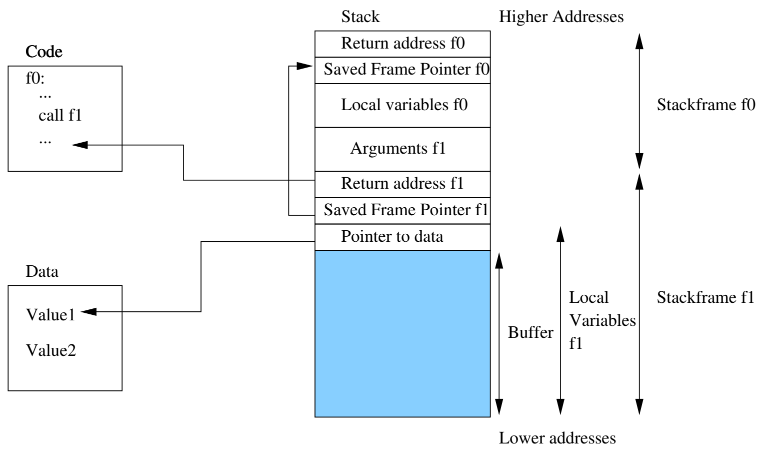


Figure 2.1: Stack-layout on the IA32-architecture

will be generated which could give attackers a second chance.

### Exploitation

**Basic exploitation** Figure 2.2 shows what could happen if attackers are able to make the program copy data beyond the end of an array (the colored part is under the attackers' control). Besides the contents of the buffer, the attackers have overwritten the pointer, the saved frame pointer (these last two have been left unchanged in this case) and the return address of the function. They could continue to write into the older stackframe if so desired, but in most cases overwriting the return address is an attacker's main objective as it is the easiest way to gain control over the program's execution-flow. The attackers have changed the return address to point to code that they copied into the buffer, probably using the same copying operation that they used to copy past the end of the buffer. When the function returns, the return address would, in normal cases, be used to resume execution after the function has ended. But since the return address of the function has been overwritten with a pointer to the attacker's injected code, execution-flow will be transfered there [3, 157].

**Frame pointer overwriting** In some cases it might not be possible for attackers to overwrite the return address. This could be because they can only overwrite a few bytes and cannot reach the return address or because the return address has been protected by some countermeasure. Figure 2.3 shows how an attacker could

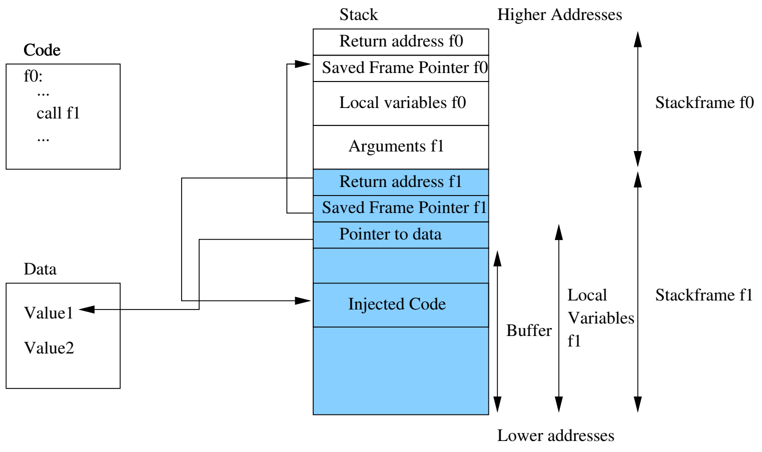


Figure 2.2: Normal stack-based buffer overflow

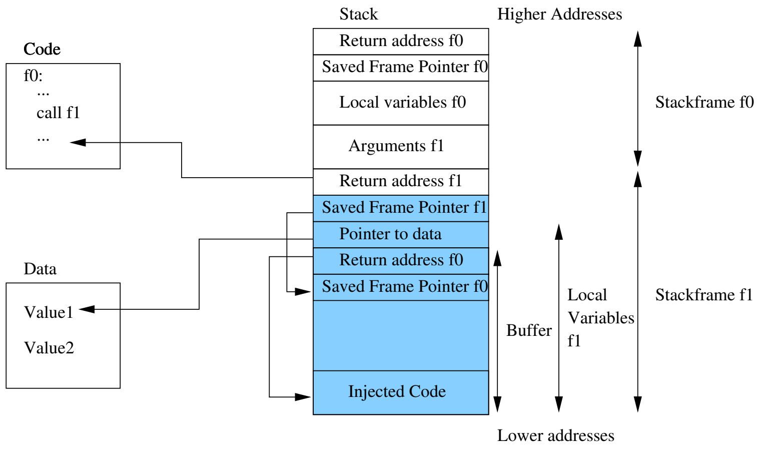


Figure 2.3: Stack-based buffer overflow overwriting frame pointer

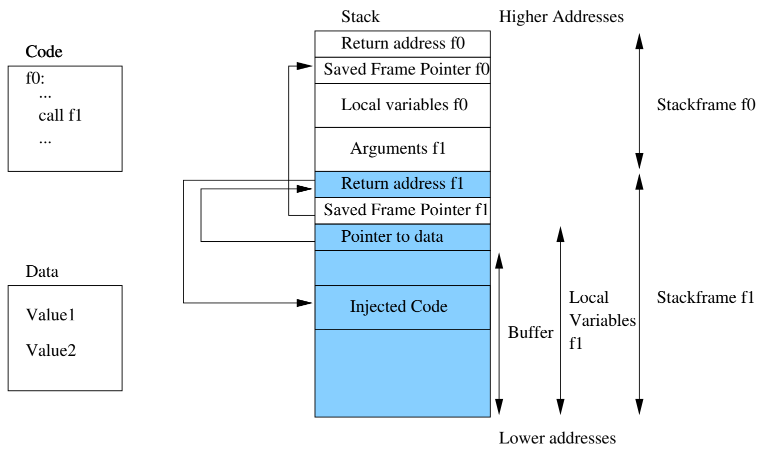


Figure 2.4: Stack-based buffer overflow using indirect pointer overwriting

manipulate the frame pointer to still be able to gain control over the execution-flow of the program: the saved frame pointer would be set to point to a different location instead of to the saved frame pointer of the previous stackframe [97]. When the function ends, the frame pointer register will be moved to the stack pointer register, effectively freeing the stack of local variables. Subsequently the old frame pointer register of $f0$ will be restored into the frame pointer register by popping the saved frame pointer off the stack. Finally the function will return by popping the return address in the instruction pointer register. In our example attackers have changed the saved frame pointer to point to a value they control instead of the frame pointer for stackframe $f0$. When the function $f1$ returns, the new saved frame pointer for $f0$ will be stored into the register which points to attacker-controlled memory. When the frame pointer register is used to 'free' the stack during $f0$'s function epilogue, the program will read the attacker-specified saved frame pointer and will transfer control to the attacker's return address when returning.

**Indirect pointer overwriting** If attackers, for some reason, cannot overwrite the return address or frame pointer directly (some countermeasures prevent this), they can use a different technique, illustrated in Figure 2.4, called indirect pointer overwriting [34], which might still allow them to gain control over the execution-flow.

The overflow is used to overwrite the local variable of $f1$ holding the pointer to value1. The pointer is changed to point to the return address instead of pointing to value1. If the pointer is then dereferenced and the value it points to is changed at some point in the function $f1$ to an attacker-specified value, then they can use it to change the return address to a value of their choosing.

Although in our example we illustrate this technique by overwriting the return address, indirect pointer overwriting can be used to overwrite arbitrary memory locations: any pointer to code that will executed later could be interesting for an attacker to overwrite.

### Heap-based buffer overflows

**Vulnerability** Heap memory is dynamically allocated at run-time by the application. As is the case with stack-based arrays, arrays on the heap can, in most implementations, be overflowed too. The technique for overflowing is the same except that the heap grows upwards in memory instead of downwards. However no return addresses are stored on the heap, so an attacker must use other techniques to gain control of the execution-flow.

### Exploitation

**Basic exploitation** One way of exploiting a buffer overflow located on the heap is by overwriting heap-stored function pointers that are located after the buffer that is being overflowed [45]. Function pointers are not always available though, so other ways of exploiting heap-based overflows must be used. For example, by overwriting a heap-allocated object's virtual function pointer [140] and pointing it to an attacker-generated virtual function table. When the application attempts to execute one of these virtual methods, it will execute the code to which the attacker-controlled pointer refers.

**Dynamic memory allocators** Function pointers or virtual function pointers are not always available when an attacker encounters a heap-based buffer overflow. Overwriting the memory management information that is generally associated with a dynamically allocated block [162, 5, 91, 13] can be a more general way of exploiting a heap-based buffer overflow.

The countermeasures we will be examining are often based on a specific implementation of a dynamic memory allocator called *dlmalloc* [107]. We will describe this allocator in short and will describe how an attacker can manipulate the application into overwriting arbitrary memory locations by overwriting the allocator's memory management information.

The *dlmalloc* library is a run-time memory allocator that divides the heap memory at its disposal into contiguous chunks, that change size as the various allocation and free routines are called. An invariant is that a free chunk never borders another free chunk when one of these routines has completed: if two free chunks had bordered, they would have been coalesced into a larger free chunk.

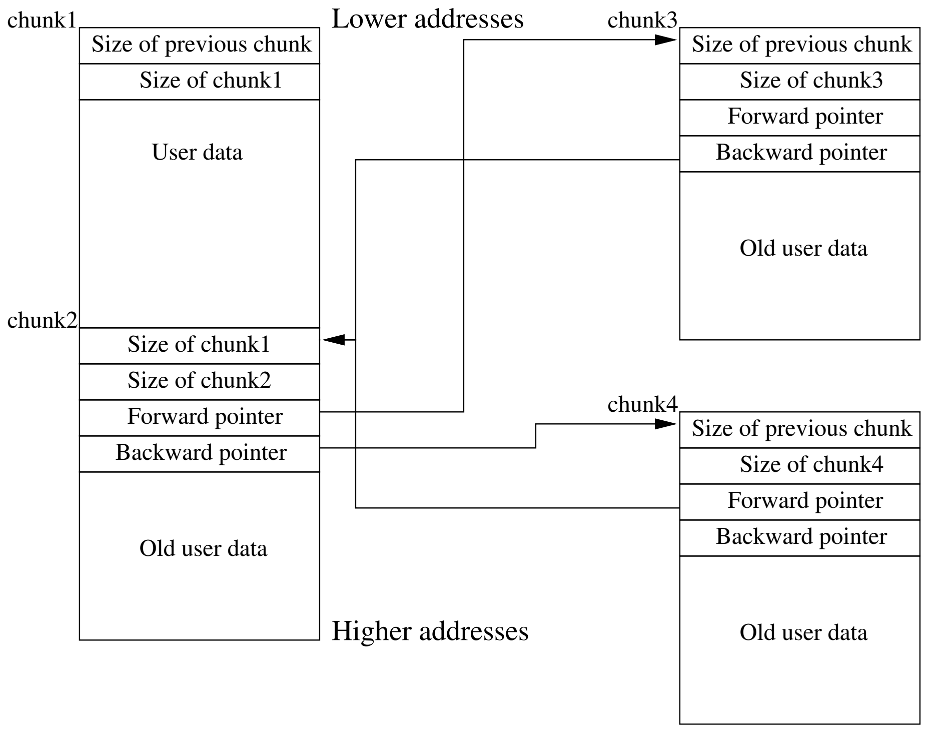


Figure 2.5: Heap containing used and free chunks

These free chunks are kept in a doubly linked list of free chunks, sorted by size. When the memory allocator at a later time requests a chunk of the same size as one of these free chunks, the first chunk in the list will be removed from the list and will be available for use in the program (i.e., it will turn into an allocated chunk).

All memory management information (including this list of free chunks) is stored in-band (i.e., the information is stored in the chunks: when a chunk is freed the memory normally allocated for data is used to store a forward and backward pointer). Figure 2.5 illustrates what a heap of used and unused chunks could look like. $Chunk1$ is an allocated chunk containing information about the size of the chunk stored before it and its own size[2]. The rest of the chunk is available for the

[2]The size of allocated chunks is always a multiple of eight, so the three least significant bits of the size field are used for management information: a bit to indicate if the previous chunk is in use or not and one to indicate if the memory is mapped or not. The last bit is currently unused. The "previous chunk in use"-bit can be modified by an attacker to force coalescing of chunks. How this coalescing can be abused is explained later.

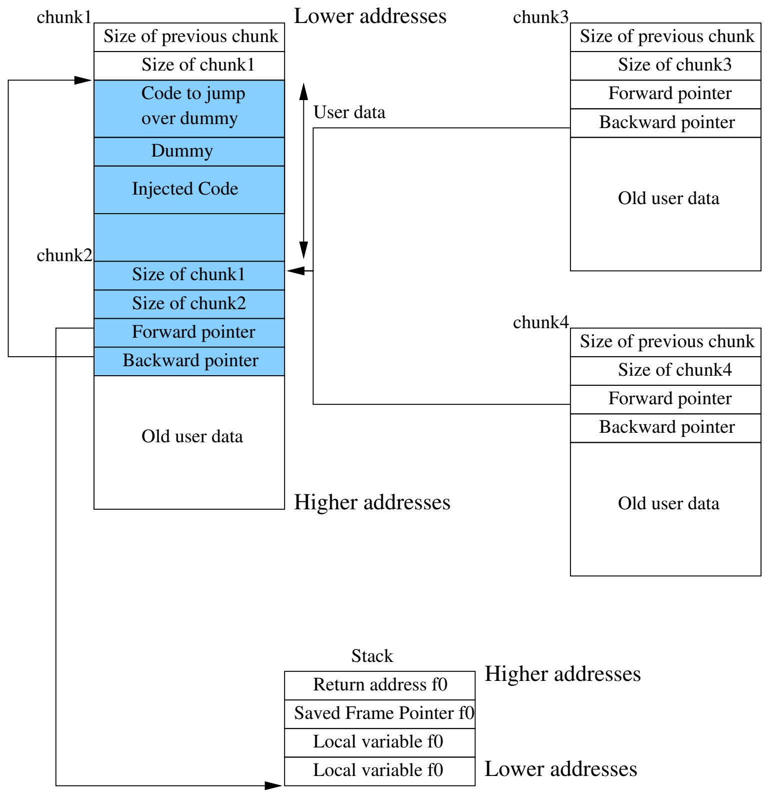


Figure 2.6: Heap-based buffer overflow

program to write data in. *Chunk*2[3] shows a free chunk that is located in a doubly linked list together with *chunk*3 and *chunk*4. *Chunk*3 is the first chunk in the chain, followed by *chunk*2 and *chunk*4, respectively.

Figure 2.6 shows what could happen if an array that is located in *chunk*1 overflows: an attacker has overwritten the management information of *chunk*2. The size fields are left unchanged in this case (although these could be modified if needed). The forward pointer has been changed to point to 12 bytes before the return address and the backward pointer has been changed to point to code that will jump over the next few bytes. When *chunk*1 is subsequently freed, it will be coalesced together with chunk2 into a larger chunk. As *chunk*2 will no longer

[3]The representation of *chunk*2 is not entirely correct: if *chunk*1 is in use, it will be used to store 'user data' for *chunk*1 and not the size of *chunk*1. We have elected to represent it this way as this detail is not relevant to the discussion.

be a separate chunk after the coalescence, it must first be removed from the list of free chunks. The unlink macro takes care of this: internally a free chunk is represented by struct containing the following unsigned long integer fields (in this order): *prev_size*, *size*, *forward* and *back*. A chunk is unlinked as follows:

Listing 2.1: Unlink macro

```
chunk2->forward->back = chunk2->back
chunk2->back->forward = chunk2->forward
```

Which is the same as (based on the struct used to represent malloc chunks):

Listing 2.2: Unlink macro expanded

```
*(chunk2->forward+12) = chunk2->back
*(chunk2->back+8) =  chunk2->forward
```

So the value of the memory location located 12 bytes after the location that *forward* points to will be overwritten with the value of *back*. And the value of the memory location 8 bytes after the location that *back* points to will be overwritten with the value of *forward*. So in the example in Figure 2.6 the return address would be overwritten with a pointer to code that will jump over the place where *forward* will be stored and will execute code that the attacker injected. As with the indirect pointer overwrite this technique can be used to overwrite arbitrary memory locations.

### 2.2.2 Dangling pointer references

#### Vulnerability

A dangling pointer reference is a pointer to a memory location that has been deallocated either explicitly by the programmer (e.g., by calling free) or by code generated by the compiler (e.g.,f a function epilogue, where the stackframe of the function is removed from the stack). Dereferencing of such a pointer is generally unchecked in a C compiler, causing the dangling pointer reference to become a problem. In normal cases this would cause the program to crash or exhibit uncontrolled behavior as the value could have been changed at any place in the program.

However, in some specific cases, if the program continues to write to memory that has been released and reused, it could also result in an attacker being able to overwrite information in a memory region that he was never supposed to write to. Even reading of such memory could result in a vulnerability where information stored in the reused memory is leaked.

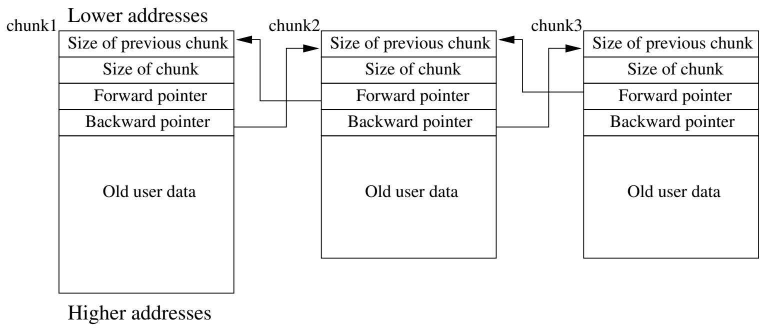


Figure 2.7: List of free chunks

If a program continues to write to memory via a dangling pointer reference after the memory has been released, it could overwrite data of an other object, which may be of a different type. For example, if a program writes to an object containing an array of characters via a dangling pointer reference, the new object which is stored there may have stored a pointer there. This means the pointer could be corrupted by the array of characters. Whether this kind vulnerability is exploitable or not, is very program specific. As such, we will focus on a specific example of such a write-after-free problem which is more generally exploitable: the double free vulnerability. A double free vulnerability occurs when already freed memory is deallocated a second time. This could again allow an attacker to overwrite arbitrary memory locations [56].

Figure 2.7 is an example of what the list of free chunks of memory might look like when using the *dlmalloc* memory allocator. *Chunk*1 is bigger than *chunk*2 and *chunk*3, meaning that *chunk*2 is the first chunk in the list of free chunks of its size. When a new chunk of the same size as *chunk*2 is freed, it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of *chunk*1 and the forward pointer of *chunk*2.

When a chunk is freed twice it will overwrite the forward and backward pointers and could allow an attacker to overwrite arbitrary memory locations at some later point in the program.

### Exploitation

As mentioned in the previous section: if a chunk (*chunk*4) of the same size as *chunk*2 is freed it will be placed before *chunk*2 in the list. The following code snippet does this:

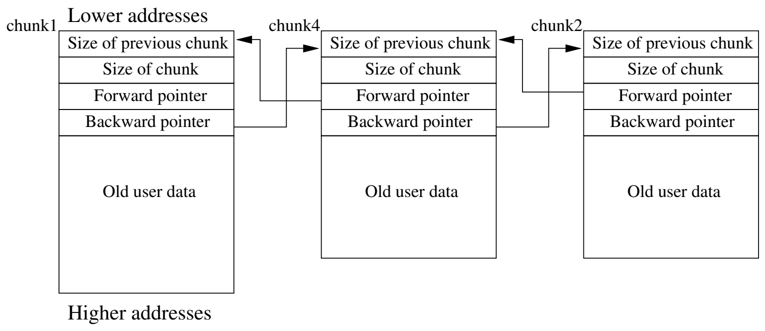


Figure 2.8: *Chunk*4 added to the list of free chunks (*chunk*3 not shown)

Listing 2.3: Adding a chunk to the list of free chunks

```
tmpback = front_of_list_of_same_size_chunks
tmpforward = tmpback->forward
chunk4->back = tmpback
chunk4->forward = tmpforward
tmpforward->back = tmpback->forward = chunk4
```

The backward pointer of *chunk*4 is set to point to *chunk*2, the forward pointer of this backward pointer (i.e., $chunk2 \rightarrow forward = chunk1$) will be set as the forward pointer for *chunk*4. The backward pointer of the forward pointer (i.e., $chunk1 \rightarrow back$) will be set to *chunk*4 and the forward pointer of the backward pointer ($chunk2 \rightarrow forward$) will be set to *chunk*4. Figure 2.8 illustrates this (*chunk*3 is not shown in this figure due to space restraints).

If chunk2 would be freed twice the following would happen (substitutions made on the code listed above):

Listing 2.4: Freeing chunk2 twice

```
tmpback = chunk2
tmpforward = chunk2->forward
chunk2->back = chunk2
chunk2->forward = chunk2->forward
chunk2->forward->back = chunk2->forward = chunk2
```

The forward and backward pointers of *chunk*2 both point to itself. Figure 2.9 illustrates what the list of free chunks looks like after a second free of *chunk*2.

If the program subsequently requests a chunk of the same size as *chunk*2 then *chunk*2 will first be unlinked from the list of free chunks:

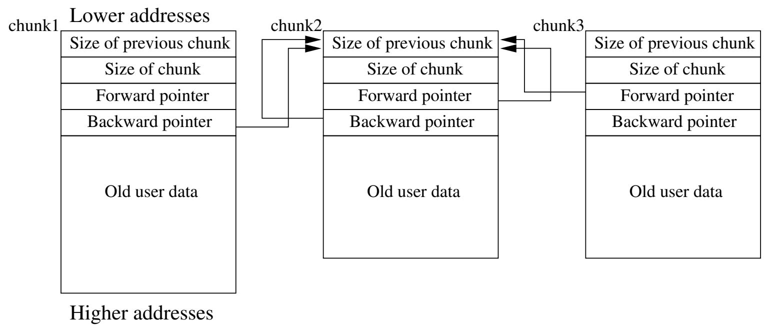


Figure 2.9: List of free chunks with chunk2 freed twice

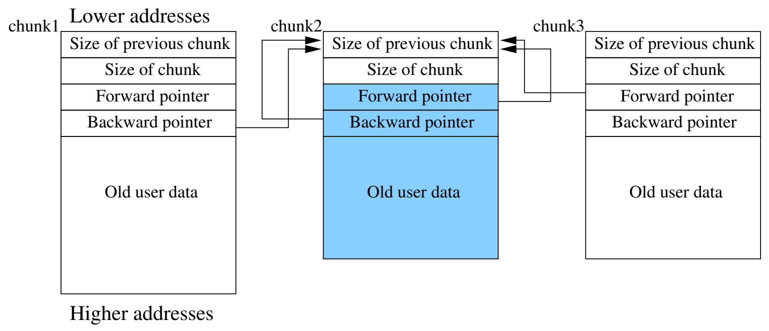


Figure 2.10: *Chunk*2 reallocated as used chunk

Listing 2.5: Unlinking chunk2

```
chunk2->forward->back = chunk2->back
chunk2->back->forward = chunk2->forward
```

But since both $chunk2 \rightarrow forward$ and $chunk2 \rightarrow back$ point to $chunk2$, it will again point to itself and will not really be unlinked. However the allocator assumes it has and the program is now free to use the user data part the chunk for its own use. Figure 2.10 illustrates where the program can now write.

Attackers can now use the same technique as was used in Section 2.2.1 to exploit the heap-based overflow (Figure 2.11): they set the forward pointer to point 12 bytes before the return address and change the value of the backward pointer to point to code that will jump over the bytes that will be overwritten. When the program tries to allocate a chunk of the same size again, it will again

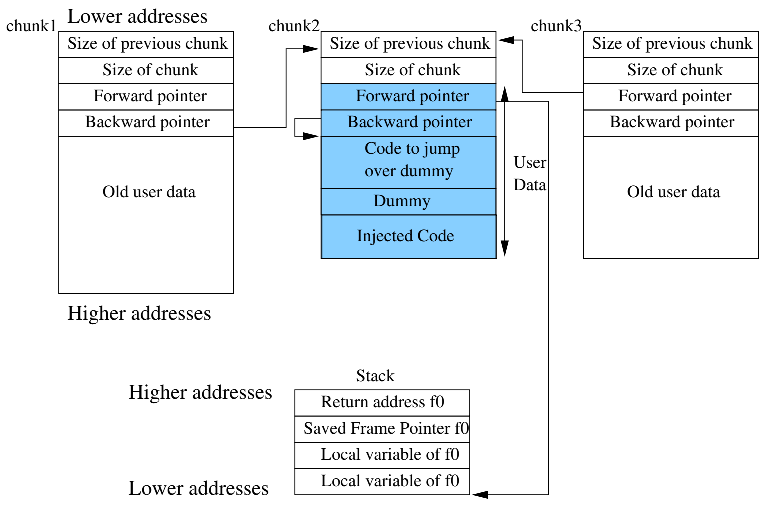


Figure 2.11: Overwriting the return address using a double free

try to unlink $chunk2$, which will overwrite the return address with the value of $chunk2's$ backward pointer.

## 2.2.3 Format string vulnerabilities

### Vulnerability

Format functions are functions that have a variable number of arguments and expect a format string as argument. This format string specifies how the format function will format its output. The format string is a character string that is literally copied to the output stream unless a % character is encountered. This character is followed by format specifiers that manipulate the way the output is generated. When a format specifier requires an argument, the format function expects to find this argument on the stack (e.g., consider the following call: $printf(''\%d'', d)$), here printf expects to find the integer d as second argument to the printf call on the stack and will read this memory location and output it to screen). A format string vulnerability occurs if an attacker is able to specify the format string to a format function (e.g., $printf(s)$, where $s$ is a user-supplied string). The attacker is now able to control what the function pops from the stack and can make the program write to arbitrary memory locations.

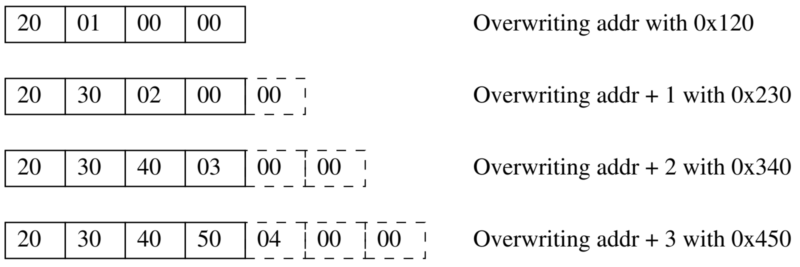

20 01 00 00
Overwriting addr with 0x120
20 30 02 00 00
Overwriting addr + 1 with 0x230
20 30 40 03 00 00
Overwriting addr + 2 with 0x340
20 30 40 50 04 00 00
Overwriting addr + 3 with 0x450

Figure 2.12: Overwriting a value 1 byte at a time using %n specifiers

**Exploitation**

One format specifier is particularly interesting to attackers: $\%n$. This specifier writes the amount of characters that have been formatted so far to a pointer that is provided as an argument to the format function [90].

Thus if attackers are able to specify the format string, they can use format specifiers like $\%x$ (print the hex value of an integer) to pop words off the stack, until they reach a pointer to a value they wish to overwrite. This value can then be overwritten by crafting a special format string with $\%n$ specifiers [147]. However addresses are usually large numbers, especially if an attacker is trying to execute code from the stack, and specifying such a large string would probably not be possible. However, format functions also accept minimum field width specifiers when reading format specifiers. The amount of bytes specified by this minimum field width will be taken into account when the $\%n$ specifier is used (e.g., $printf(''\%08x'', d)$ prints $d$ as an eight digit hexadecimal number: if $d$ has the decimal value 10 it would be printed as $0000000a$). This field width specifier makes it easier to specify a large format string but the number attackers are required to generate will still be too large to effectively be used. To overcome this limitation, they can write the value in four parts: overwriting the return address with a small value (normal integers on the IA32 overwrite four bytes), then overwriting the return address + one byte with another integer, then return address + two bytes and finally return address + three bytes.

The attacker faces one last problem: the amount of characters that have been formatted so far is not reset when a $\%n$ specifier is written. If the address the attackers want to write contains a number smaller than the current value of the $\%n$ specifier this could cause problems. But since the attackers are writing one byte at a time using a four-byte value, they can write larger values with the same least significant byte (e.g., if attackers want to write the value $0x20$, they could just as well write $0x120$). Figure 2.12 illustrates how an attacker would write $0x50403020$ in an address using this technique.

### 2.2.4 Integer errors

Integer errors [26, 195] are not exploitable vulnerabilities by themselves, but exploitation of these errors could lead to a situation where the program becomes vulnerable to one of the previously described vulnerabilities. Two kinds of integer errors that can lead to exploitable vulnerabilities exist: integer overflows and integer signedness errors.

- An integer overflow occurs when an integer grows larger than the value that it can hold. The ISO C99 standard [90] mandates that unsigned integers that overflow must have a modulo of $MAXINT + 1$ performed on them and the new value must be stored. If an integer overflows, the resulting modulo will make it wrap around 0 (i.e., 65536 + 2, will store the value 1 in a unsigned 16 bit integer). This can cause a program that does not expect this to fail or become vulnerable: if used in conjunction with memory allocation, too little memory might be allocated causing a possible heap overflow. For example, if a program uses the integer together with an addition to make sure that enough memory is allocated to hold a copy of a specific array, an integer overflow could cause the program to allocate too little memory because the value has wrapped around 0.

- Integer signedness errors on the other hand are more subtle: when the programmer defines an integer, it is assumed to be a signed integer, unless explicitly declared unsigned. When the programmer later passes this integer as an argument to a function expecting an unsigned value, an implicit cast will occur. This can lead to a situation where a negative argument passes a maximum size test but is used as a large unsigned value afterwards, possibly causing a stack or heap overflow if used in conjunction with a copy operation (e.g., *memcpy*[4] expects an unsigned integer as size argument and when passed a negative signed integer, it assumes this is a large unsigned value).

Integer errors can cause problems for static analyzers that check for buffer overflows, because they assume the existence of "infinite precision arithmetic". This means that they will not take into account that an integer could overflow. This limitation can cause false negatives for static analyzers which are otherwise considered sound.

## 2.3 Countermeasure properties

This survey aims to provide an understanding of the field of code injection attacks to software engineers and computer scientists without specific security expertise.

[4]*memcpy* is the standard C library function that is used to copy memory from one location to another

| Code | Type |
|---|---|
| *D* | Detection: Attacks on vulnerabilities are detected (i.e. after memory has been corrupted) |
| *P* | Prevention: The vulnerability is prevented and possibly detected (i.e. before memory is corrupted) |
| *M* | Mitigation: Exploitation is made harder, no explicit detection |
| *C* | Containment: Limits the damage of exploitation |

Table 2.1: Countermeasure type

We do this by providing a structured classification and evaluation framework of countermeasures that exist to deal with these types of attacks. At the top level, we classify existing countermeasures based on the main technique they use to address the problem.

However, countermeasures also make different trade-offs in terms of performance, effectiveness, memory cost, compatibility, etc.. In this section we define a number of properties that can be attributed to each countermeasure. Based on these properties advantages and disadvantages of different countermeasures can be assessed.

### 2.3.1 Type

The types of protection that countermeasures provide are contained in Table 2.1. Countermeasures that offer detection detect an attack when it occurs and take action to defend the application against it, but do not prevent the vulnerability from occurring. Prevention countermeasures attempt to prevent the vulnerability from existing in the first place. As such, they are generally not able to detect when an attacker is attempting to exploit a program as the vulnerability should have been eliminated. Countermeasures that make it harder for an attacker to exploit a vulnerability but that don't actually detect an exploitation attempt are of the type mitigation. Finally the last type of countermeasures are the ones that do not try to detect, prevent or mitigate an attack or a vulnerability but try to contain the damage that an attacker can do after exploiting a vulnerability.

### 2.3.2 Vulnerabilities

Table 2.2 contains a list of the vulnerabilities that the countermeasures in this survey deal with. They reflect the scope of the vulnerabilities that the designer of the countermeasure wished to address. However, in some cases a countermeasure implicitly offers protection against exploitation for a certain kind of vulnerability without explicitly being designed to protect against it (this is especially true for integer errors) . In such cases we have also listed these vulnerabilities as being in

| Code | Vulnerability |
|---|---|
| *S* | Stack-based buffer overflow |
| *H* | Heap-based buffer overflow |
| *D* | Dangling pointer references |
| *F* | Format string vulnerabilities |
| *I* | Integer errors |

Table 2.2: Vulnerability addressed by a countermeasure

| Code | Protection level |
|---|---|
| *L* | Low assurance |
| *M* | Medium assurance |
| *H* | High assurance |

Table 2.3: Protection level of a countermeasure

the scope of the countermeasure. For example, some bounds checking solutions offer protection against exploitation of integer errors because integer errors are usually abused to cause buffer overflows. Thus we have listed exploitation of integer errors as being part of the protection offered by bounds checking solutions, even though they do not offer explicit protection.

### 2.3.3 Protection level

This property describes the level of protection a countermeasure provides for the vulnerabilities it was designed for.

#### Low assurance countermeasures

Low assurance countermeasures make exploiting a vulnerability harder, however a method to bypass this countermeasure has been discovered and is practical. For example, a return-into-libc attack is a practical attack on non-executable memory countermeasures [188, 150].

#### Medium assurance countermeasures

Medium assurance countermeasures offer better protection than low assurance countermeasures: as long as the assumptions that the countermeasure was built on are preserved, no way of bypassing these countermeasures is currently known that is practical. However, these assumptions do not always hold in the real world. There may also be a way of bypassing these countermeasures which is not immediately practical, but which may be possible. An example of countermeasures that fall under this category are canary-based countermeasures that use random

| Code | Stage |
|---|---|
| *Imp* | Implementation |
| *Test* | Debugging & Testing |
| *Pack* | Packaging |
| *Depl* | Deployment |

Table 2.4: Stage of the software engineering process that a countermeasure is applied at

numbers for protection. The random number must remain secret, which is an assumption that does not always hold in the real world: attackers may be able to find out the number through memory leaks. An attack may also be able to guess the random number given enough attempts, even though the range of possibilities may be too high to make this immediately practical (e.g. some countermeasures have $2^32$ possible combinations on a 32-bit systems).

**High assurance countermeasures**

High assurance countermeasures offer a high degree of assurance that they will work against a specific vulnerability (e.g. if a countermeasure only targets buffer overflows and has high assurance, it will only have high assurance for buffer overflows). Countermeasures which offer memory safety, type safety or can offer verifiable guarantees will have a high assurance rating. For example, a safe language that explicitly removes or checks constructs to prevent a specific vulnerability from occurring will have a high assurance protection level. Sometimes weaker countermeasures which still offer a high level of assurance will also fall in this section. For example, a bounds checker which ensures that no object writes outside its bounds will have a high assurance protection level even though it may be possible for an attacker to overwrite a pointer inside a structure via another element in the same structure (something many bounds checker will not detect because this is valid according to the C standard: preventing this type of vulnerability would break valid programs).

Static analyzers that do not suffer from false negatives for a given vulnerability and set of assumptions (i.e. sound analyzers) will also fall under this category, since they will alert the programmer of all occurrences of that vulnerability.

## 2.3.4 Usability

This property describes how the countermeasures can be applied. We differentiate between two subproperties of usability: stage and effort.

Stage (Table 2.4) denotes where in the software engineering process the countermeasure can be applied.

| Code | Effort |
|---|---|
| *Auto* | Automatic |
| *ManS* | Small manual effort |
| *ManL* | Larger manual effort |

Table 2.5: Effort required to apply a countermeasure

None of the countermeasures in this survey operate on the requirement, analysis or design stages of a product, so these stages have been left out of the table. Countermeasures that affect the way an application is coded (i.e., safe languages and static analysis tools) reside in the implementation stage. Tools that do dynamic analysis fall under the debugging and testing stage. Some countermeasures are compiled into the program, or modify the binary before it is shipped to customers. These countermeasures operate at the packaging stage. Deployment countermeasures are only applied after the program has been shipped to the customer and usually protect more than one application (e.g., kernel patches, sandboxes, etc.).

Effort (Table 2.5) describes the amount of effort required to use the countermeasure. We define a countermeasure to be automatic if it requires no further human effort besides applying the countermeasure. Manual countermeasure requires more effort for a countermeasure to be applied (e.g., annotations, modification of source code). We also apply a modifier to quantify the amount of manual effort required, small or large.

The effort properties should be viewed in combination with the way the countermeasure is applied. A static countermeasure adds no code to protect against a vulnerability but can never be fully automatic because it requires human intervention to interpret the results (e.g., to fix the reported vulnerability). As such, if we denote a static countermeasure as automatic, if it does not require annotations or other help from the person running the analysis.

### 2.3.5 Limitations

In Table 2.6 we list the category of limitations in applicability of a countermeasure. Some countermeasures are implemented as hardware changes, this can be a limiting factor in being able to apply a countermeasure. Other countermeasures are implemented as modifications to the operating system. This is not as major a limiting factor for applying a countermeasure as requiring hardware changes. In some cases applying a countermeasure at the OS-level can even be a benefit: an OS-based countermeasure should work for all software running on the OS. Some countermeasures also require access to the source code or at least to the debugging symbols, so that they can instrument the software that is being protected correctly. In some cases, countermeasures will be incompatible with existing compiled code like libraries. This is especially the case if they modify binary representations of

| Code | Limitations |
|---|---|
| *HW* | Hardware (or virtual machine) changes needed |
| *OS* | Operating system changes needed |
| *Src* | Source code required |
| *Dyn* | Dynamically linked executable required |
| *Deb* | Debugging symbols required |
| *Inc* | Incompatible with existing compiled code |
| *Chg* | Possible changes required in source code |
| *Arch* | Architecture or operating system specific |
| *False* | May suffer from false positives (identifies a program as vulnerable when it's not) |

Table 2.6: Limitations of a countermeasure

particular datatypes (like pointers). Safe languages often require a programmer to modify his source code so that it can become easier to analyze the code and apply countermeasures. A few of the countermeasures described in this survey rely on specific features of some architectures or operating systems. This makes it unlikely that they could be ported to the other architectures without significant reengineering. A number of countermeasures also suffer from false positives and/or false negatives. However, if the countermeasure does not offer high assurance, it suffers from false negatives. As such, we have added a property to denote whether false positives are present or not to the limitations.

### 2.3.6 Computational and memory cost

The computational and memory costs give an estimate of the run-time cost a specific countermeasure could incur when deployed. The values listed there are provided as-is, in some cases it was extremely hard to determine the cost based on the descriptions given by the authors and as such some values in these columns might not be entirely accurate. The costs range from none to very high for both computational and memory cost.

For dynamic countermeasures the costs are measured as a comparison to running non-instrumented programs. The costs of these countermeasures cannot be compared to the costs of static analysis. Costs for countermeasures that use static analysis are measured in running time of the analysis.

## 2.4 Countermeasures

In this section we provide a description of the different categories of countermeasures and an overview of the properties of specific countermeasures. At the top

| Code | Cost |
|---|---|
| ? | Unknown |
| 0 | None |
| −− | Very low |
| − | Low |
| −+ | Medium |
| + | High |
| ++ | Very high |

Table 2.7: Computational and memory cost of a countermeasure

level, we distinguish between ten categories based on the main technique that was used to design the countermeasure. Each of these categories is discussed in a separate subsection. We first describe the key ideas behind the category, and the main advantages and disadvantages. Next, we provide a table listing all proposed countermeasures in that category. The table evaluates each of the countermeasures by providing values for each of the properties discussed in the previous section.

Categories may overlap, i.e., some countermeasures could be placed in one category or in another. This is because many countermeasures will use a variety of techniques to provide protection. When this was the case, we chose the most important technique or the most novel technique discussed in the countermeasures' paper to determine the category to place the countermeasure in.

For some categories of countermeasures we describe specific subtechniques. For example, within the static analysis category, we discuss various techniques on which a static analyzer can be based. When such techniques are discussed, an extra column "Tech" is added to the table of properties to denote the specific technique that was used to implement specific countermeasures.

Finally, for some categories, all countermeasures have the same value for all countermeasures in that category. When this is the case, the column for that property is dropped from the table for clarity. The description of the category always notes if a column was dropped and shows which value it would have contained.

### 2.4.1 Safe languages

Safe languages are languages in which it is generally not possible for one of the vulnerabilities discussed in Section 2.2 to exist as the language constructs prevent them from occurring (this means they could be memory safe, type safe or neither). A number of safe languages are available that prevent the kinds of implementation vulnerabilities discussed in this text. Examples of such languages include Java and ML, but these are not in the scope of our discussion since this survey focuses on C and C++. Thus we will only discuss safe languages which remain as close to C or C++ as possible: these safe languages are mostly referred to as safe dialects of

C. Some dialects [121] will only need minimal programmer intervention to compile programs, while others [87, 116] require substantial modification. Others [98] severely restrict the C language to a subset to make it safer or will prevent behavior that the C standard marks as undefined [124].

In an attempt to prevent dangling pointer references, memory management is handled differently by these safe languages: in some cases the programmer is not given explicit control over deallocation anymore (i.e., the free operation is either replaced with a no-operation or removed altogether). In the languages listed in Table 2.8 three types of memory management are used to prevent dangling pointer references:

**Garbage collection** does not deallocate memory instantaneously, but defers this to a scheduled time interval or till memory constraints require it to be collected. When garbage collection is done, only memory to which no references exist is deallocated, preventing pointers from referring to deallocated memory [29]. An important problem with using garbage collection for C programs, however is that C programs will generally not clear all pointers to a memory location when they free that location. As such, using garbage collection without modifying the program could result in the program using an unacceptable amount of memory.

**Region-based memory management** deallocates regions as a whole, memory locations can no longer be deallocated separately. A pointer to a memory location can only be dereferenced if the region that this memory location resides in is marked ”live”. Programmers can manually allocate memory in such a region but can only deallocated the region as a whole. This introduces some problems with objects that live too long, as a result of being placed in a region that remains live very long. As such this type of memory management usually requires garbage collection, to deallocate heap-based objects (which are in one large region). However by ensuring that a pointer is unique upon deallocation (i.e., the pointer has no aliases), a programmer can safely deallocate memory without causing dangling pointer references [80].

**Automatic Pool Allocation** makes use of a points-to graph to allocate objects that have the same node in the points-to graph in a same pool in the heap. As a result, all objects of the same type are allocated in the same pool. This allows programmers to manually allocate and free memory in the heap. While dangling pointer references may occur, they point to the same type of object in memory. As a result, it is harder to break memory safety when Automatic Pool Allocation is used [106].

To prevent the remaining implementation errors that we described in Section 2.2, several techniques are usually combined: firstly static analysis can be performed to determine if a specific construct can be proven safe. If the construct cannot be proven safe, then run-time checks are added to prevent errors at run-time (e.g., if use of a specific array cannot statically be determined to be safe then code that does run-time bounds checking might be added). To aid this static analysis, pointers are often divided into different types (e.g., never-null pointers that are guaranteed to never contain a NULL value) depending on the amount of checking that should be added at runtime. Some of the safe languages infer these pointer types automatically, while others expect the programmer to explicitly use new types of pointers.

| Name | Type | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|
| Cyclone [87] | P | All | H | Imp | ManL | Src +Chg | $-+/-+$ |
| Ccured [121] | P | All | H | Imp | ManS | Src + Chg | +/+ |
| Vault [105] | P | All | H | Imp | ManL | Src + Chg | ?/? |
| Control-C [55, 98] | P | All | H | Imp | ManL | Src + Chg | ?/? |
| Fail-Safe C [124] | P | All | H | Pack | ManS | Src + Chg | +/+ |
| Xu et al. [193] | P | SHDI | H | Pack | Auto | Src + Chg | +/+ |
| SAFECode [54] | P | SHD | H | Pack | ManS | Src + Chg | $-+/+$ |
| Deputy [43] | P | SH | H | Imp | ManS | Src + Chg | $-+/+$ |

Table 2.8: Safe languages

Table 2.8 gives an overview of the safe languages we examined. Some global properties can be gathered from the table: none of the languages support all constructs in C: some programs will need to be modified. Though the effort required to implement these changes varies,this is reflected in the effort property in Table 2.8.

Most languages work at the implementation level: the programmer either implements his program directly for the language or modifies his program to make it work. Fail-safe C [124] on the other hand is not a new language, it is a C compiler that attempts to make C safer by preventing behavior which is listed as undefined by the C standard. As a result, it will not compile all programs because many programs will rely on a specific compiler-dependent behavior of a compiler. For example, the C standard lists creating a pointer which points more than one element past the end of an array as undefined behavior, however many programs will expect to be able to do this. This was evidenced by the need for an extension to the Jones & Kelly bounds checker, to allow programs to create such out of bounds pointers [144].

Control-C [55, 98] is also of particular interest because it does not add dynamic checks when compiling a program: it restricts the C language to a specific subset and makes a number of assumptions about the runtime system (it is designed to run on the Low Level Virtual Machine (LLVM) system, where all memory and register operations are type safe).

The computational and memory costs of these languages are inversely related to the amount of effort required to port a program to the language: small effort means higher overheads while larger effort means lower overheads.

### 2.4.2 Bounds checkers

Bounds checkers provide extensive protection against exploitation of buffer overflows: they check array indexation and pointer arithmetic to ensure that they do not attempt to write to or read from a location outside of the space allocated for them.

Two important techniques are used to perform traditional full bounds checking:

- Adding bounds information to all pointers. These bounds checkers add extra information to all pointers: besides the current value of the pointer, the lower and upper bound of the object the pointer refers are also stored. This information can be stored in two ways: by changing the representation of the pointer itself to include all this information (these are called fat pointers) or by storing the extra information in a table for each pointer and to look it up when needed. When the pointer is used, a check is performed to make sure it does not write beyond the bounds of the object it refers to. A problem with fat pointers is that they are incompatible with existing code that was compiled with another compiler: fat pointers must be cast back and forth. If a pointer is modified by unchecked code (e.g., if it is a global variable) it may point into a new object. When it is later checked, it is found to be out of bounds, because the bounds information associated with it still refers to the original object.

- Adding bounds information for objects. Pointers remain the same, but a table stores the bounds information of all objects. Using the pointer's value, it can be determined what object it is pointing to. Then, pointer arithmetic is checked: the bounds of the object the pointer refers to is looked up and if the result of pointer arithmetic would make the pointer point outside the bounds of the object, an overflow has been detected. Pointer use is also checked to make sure that the pointer points to a valid object.

While the previous two techniques are used to perform the traditional bounds checking, this category also contains other bounds checkers which do some kind of bounds checking but are different with respect to the traditional checkers in that they do not strive for checking of all objects. Some of the latter countermeasures ensure that a string manipulation function only writes inside the stack frame that the destination pointer is pointing to [11, 159] or ensure that the function does not write past the bounds of the destination string [10].

RICH [32] is another special kind of bounds checker: it is not a bounds checker in the traditional sense. It does not check bounds of arrays or pointers but makes sure that integers do not overflow. This type of checking can prevent integer errors from occurring. However since the C standard specifically allows integer overflows, the countermeasure suffers from false positives: it can flag a problem when correct and safe code that relies on integer overflows is executed.

| Name | Type | Tech | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|---|
| Bcc [94] | P | Ptr Chk | SHI | H | Pack | Auto | Src | +/+ |
| RTCC [165] | P | Ptr Chk | SHI | H | Pack | ManS | Src + Chg | +/+ |
| Safe C[8] | P | Ptr Chk | SHDI | H | Pack | Auto | Src + Inc | +/+ |
| Jones et al. [89] | P | Obj Chk | SHI | H | Pack | ManS | Src + Chg | + + / + + |
| Suffield [168] | P | Obj Chk | SHI | H | Pack | ManS | Src + Chg | + + / + + |
| Lhee et al. [110] | P | Obj Chk | SHI | M | Pack | Auto | Src + Libc | − + /+ |
| CRED [144] | P | Obj Chk | SHI | H | Pack | Auto | Src | + + / + + |
| Rinard et al. [138] | P | Obj Chk | SHI | H | Pack | Auto | Src | + + / + + |
| Dhurjati et al. [52] | P | Obj Chk | SHI | H | Pack | Auto | Src | − + /? |
| Guarding [128] | P | Ptr Chk | SHDI | H | Pack | Auto | Src + Chg | + + / + + |
| Nethercote et al. [122] | P | Obj Chk | SHDI | L | Depl | Auto | Dyn + Deb | + + / + + |
| Cash [101] | D | Ptr Chk | SHI | A | Pack + Depl | Auto | Src + Arch + OS | −/ − + |
| Rinard et al. [137] | P | Obj Chk | SHI | H | Pack | Auto | Src | ++/+ |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Libsafe-Plus [10] | P | Other | SHI | M | Depl | Auto | Deb + Dyn + Libc | − + / − + |
| Libsafe [11] | D | Other | S | L | Depl | Auto | Dyn + Libc | −/− |
| Snarskii [159] | D | Other | S | L | Depl | Auto | Dyn + Libc | −/− |
| Healers [64] | D | Other | H | L | Depl | Auto | Dyn + Libc | +/+ |
| RICH [32] | D | Other | I | M | Pack | Auto | Src + False + Chg | − − / − − |

Table 2.9: Bounds Checkers

From Table 2.9 it is clear that most traditional full bounds checkers that offer high assurance suffer from a high computational and memory overhead.

### 2.4.3 Probabilistic countermeasures

Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures use a secret random number that are stored before an important memory location: if the random number has changed after some operations have been performed then an attack has been detected. Memory-obfuscation countermeasures encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers encrypt the instructions while in memory and will decrypt them before execution,

#### Canaries

The observation that attackers usually try to overwrite the return address when exploiting a buffer overflow led to a sequence of countermeasures that were designed to protect the return address. One of the earliest examples of this type of protection is the canary-based countermeasure [49]. This type of countermeasure protects the return address by placing a value below it on the stack that must remain unchanged during program execution. Upon entering a function the canary is placed on the stack below the return address. When the function returns, the canary stored on the stack will be compared to the original canary. If the stack-stored canary has changed an overflow has occurred and the program can be safely terminated. A canary can be a random number, or a string that is hard to replicate when exploiting a buffer overflow (e.g., a NULL byte). StackGuard [47, 49] was the first countermeasure to use canaries to offer protection against stack-based buffer overflows. However, attackers soon discovered a way of bypassing it using indirect pointer overwriting. Attackers would overwrite a local pointer in a function and make it point to a target location. When the local pointer is dereferenced for writing, the target location is overwritten without modifying the canary (see Section 2.2.1 for a more detailed description). Propolice [62] is an extension of StackGuard: it fixes these type of attacks by reordering the stack frame so that buffers can no longer overwrite pointers in a function. These two countermeasures have been extremely popular: Propolice has been integrated into the GNU C Compiler and a similar countermeasure has made its way into Visual Studio's compiler [30, 73].

Canaries were later also used to protect other memory locations, like the management information of the memory allocator that is often overwritten using a heap-based buffer overflow [100].

### Obfuscation of memory addresses

Memory-obfuscation countermeasures use an approach that is closely related to canaries: their approach is also based on random numbers. These random numbers are used to 'encrypt' specific data in memory and to decrypt it before using it in an execution. These approaches are currently used for obfuscating pointers (XOR with a secret random value) while in memory [48]. When the pointer is later used in an instruction it is first decrypted in a register (the decrypted value is never stored in memory). If an attacker attempts to overwrite the pointer with a new value, it will have the wrong value when decrypted. This will most likely cause the program to crash. A problem with this approach is that XOR encryption is bytewise encryption. If an attacker only needs to overwrite 1 or 2 bytes instead of the entire pointer, then the chances of guessing the pointer correctly vastly improve (from 1 in 4 billion to 1 in 65000) [4]. If the attacker is able to control a relatively large amount of memory (e.g., with a buffer overflow), then the chances of a successful attack increase even more. While it is possible to use better encryption, it would likely be prohibitively expensive since every pointer needs to be encrypted and decrypted this way. The idea of encrypting data in memory was later extended in [19].

### Address Space Layout Randomization

ASLR is another approach that makes executing injected code harder. Most exploits expect the memory segments to always start at a specific known address. They will attempt to overwrite the return address of a function, or some other interesting address with an address that points into their own code. However for attackers to be able to point to their own code, they must know where in memory their code resides. If the base address is generated randomly when the program is executed, it is harder for the exploit to direct the execution-flow to its injected code because it does not know the address at which the injected code is loaded. Shacham et al. [151] examine limitations to the amount of randomness that such an approach can use [5]. Their paper also describes a guessing attack that can be used against programs that use forking as the forked applications are usually not rerandomized, which could allow an attacker to keep guessing by causing forks and then trying until the address is found.

### Instruction set randomization

ISR is another technique that can be used to prevent the injection of attacker-specified code. Instruction set randomization prevents an attacker from injecting any foreign code into the application by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. Attackers are unable to guess the decryption key of the current process, so their instructions, after they've been decrypted, cause the wrong instructions to be executed. This prevents attackers from having the process execute their payload and has a large chance crashing the process due to an invalid instruction being executed. However if attackers are able to print out specific locations in memory, they can bypass the countermeasure since the encryption key can often be derived from encrypted data (since most countermeasures will use XOR). Other attacks are described in [180]. Two implementations (by Barrantes et al. and Kc et al.) [12, 93] examined in this survey incur a significant run-time performance penalty when unscrambling instructions because they are implemented in emulators, but it is entirely possible, and in most cases desirable, to implement them at the hardware level thus reducing the impact on run-time performance.

| Name | Type | Tech | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|---|
| Stack-Guard [49] | D | Can | S | L | Pack | Auto | Src | −/− |

[5] This limitation is due to address space limitations in 32-bit architectures: often countermeasure will limit randomness to a maximum amount of bits, which will be less than 32 bit, making guessing attacks a possibility.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bray et al. [30] | D | Can | S | M | Pack | Auto | Src | $-/-$ |
| Propolice [62] | D | Can | S | M | Pack | Auto | Src | $-/-$ |
| Robertson et al. [142] | D | Obfus | H | A | Depl | Auto | Dyn | $-/-$ |
| Contra-police [100] | D | Can | H | A | Depl | Auto | Dyn + Libc | $-/-$ |
| Stack-Ghost [67] | D | Obfus | S | L | Depl | Auto | Arch | $-/-$ |
| Point-Guard [48] | M | Obfus | All | L | Pack | Auto | Src | $-+/-$ |
| Zhu et al. [205] | D | Obfus | All | L | Pack | Auto | Src | $-/-$ |
| HSAP [153] | D | Bchk / Obfus | SH | L | Pack | Auto | HW + Chg | $--/0$ |
| Hu et al. [84] | D | ISR | All | M | Depl | Auto | HW | $-+/++$ |
| Barrantes et al.[12] | M | ISR | All | L | Depl | Auto | HW | $+/+$ |
| Kc et al. [93] | M | ISR | All | L | Depl | Auto | HW | $-/+$ |
| PaX ASLR [170] | M | Mem Rand | All | L | Depl | Auto | OS | $--/-$ |
| TRR [190] | M | Mem Rand | All | L | Depl | Auto | - | $-/-+$ |
| Chew et al. [41] | M | Mem Rand | S | L | Depl | Auto | OS | $-/-$ |
| Bhatkar et al. [18] | M | Mem Rand | All | L | Depl | Auto | - | $0/-$ |
| DieHard [14] | D | Mem Rand | H | M | Depl | Auto | Dyn | $-+/-+$ |
| Bhatkar et al. [20] | M | Mem Rand | All | M | Pack + Depl | Auto | Src | $-/0$ |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bhatkar et al. [19] | M | Obfus | All | M | Pack | Auto | Src | +/? |

Table 2.10: Probabilistic countermeasures

None of these countermeasures can offer a complete protection against the code injection attacks described in Section 2.2. They all rely on the fact that memory must remain secret. If an attacker is able to read out memory (through a format string vulnerability or another type of attack), then the countermeasures can be bypassed entirely. However, a major advantage of these approaches is that they have low computational and memory overheads, making them better suited for production environments. A notable exception to the previously described limitation that memory must remain secret is the work by Hu et al. [84]. In this paper an approach to instruction set randomization is described that makes use of AES encryption[6] [51] for instructions. This can reduce the risk that is posed by an attacker being able to read memory locations and finding out the key. Another advantage of the stronger encryption is that the countermeasure knows for sure if an instruction is correct or not, rather than relying on it being invalid if the attacker does not know the key.

### 2.4.4 Separation and replication of information

Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information or will separate this information from regular data. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data.

### 2.4.5 Paging-based countermeasures

Paging-based countermeasures make use of the Virtual Memory Manager, which is present in most modern architectures. Memory is grouped in contiguous regions of fixed sizes (4Kb on Intel IA32) called pages. Virtual memory is an abstraction above the physical memory pages that are present in a computer system. It allows a system to address memory pages as if they are contiguous, even if they are

[6]Advanced Encryption Standard a.k.a Rijndael is a block cipher

| Name | Type | Vulns | Assur. | Stage | Effort | Lim. | Cost |
|---|---|---|---|---|---|---|---|
| StackShield [173] | D/M | S | L | Pack | Auto | Src | $-/-$ |
| RAD [42] | D | S | L | Pack | Auto | Src | $-+/--$ |
| Xu et al. [191] | M/D | S | L | Depl | Auto | Src / HW | $--/--$ |
| SmashGuard [127] | D | S | L | Depl | Auto | HW | $--/--$ |
| Lee et al. [109] | D | S | L | Depl | Auto | HW | $--/--$ |
| Libverify [11] | D | S | L | Depl | Auto | Dyn | $-+/-$ |
| Libparanoia [158] | D | S | L | Depl | Auto | Dyn + Libc | $-/-$ |
| dnmalloc [200] | M | HD | M | Depl | Auto | Dyn | $--/-+$ |
| multistack [201] | M | S | M | Depl | Auto | Src | $--/-+$ |
| Prasad et al. [133] | D | S | L | Depl | Auto | - | $--/--$ |
| Smirnov et al. [156] | D | S | L | Pack | Auto | Src | $--/?$ |

Table 2.11: Separation Countermeasures

stored on physical memory pages that are not. An example of this is the fact that every process in Linux starts at the same address in the virtual address space, even though physically this is not the case. Another advantage of virtual memory is the fact that all application seemingly have 4GB of RAM (on 32 bit systems) available, even if the machine does not have that much physical RAM available. This also allows for the concept of swapping, where memory is written to disk when it is not in active use so the physical memory can be reused for active applications. Translation of virtual memory to physical memory is handled by a memory management unit (MMU), which is present in most architectures.

Pages can have specific permissions assigned to them: Read, Write and Execute. Many of the countermeasures in this section will make use of paging permissions or the fact that multiple virtual pages can be mapped onto the same physical page.

Countermeasures in this category can be divided into two subcategories:

**Non-executable memory (NX)** makes data memory non-executable. Most operating systems divide process memory into at least a code (also called the text) and data segment. The code segment is marked as read-only, preventing a program from modifying code that has been loaded from disk into this segment (unless the program explicitly requests write permissions for the memory region). Because of this, attackers have to inject their code into the data segment of the application. As most applications do not require executable data segments (because all their code is in the code segment), some countermeasures mark this memory as non-executable, which will make it harder for an attacker to inject code into a running application. A major disadvantage of this approach is that an attacker could use a code injection attack to execute existing code. One type of such an attack is called a return-into-libc attack [188, 150]. Instead of injecting code on the stack and then pointing the return address to this code, the desired parameters are placed on the stack and the return address is pointed to existing code. For example, if attackers use the return address to return to the libc wrapper for the *system()* system call and pass a string containing the location of an executable as argument. This allows the attackers to execute arbitrary code without injecting actual code into the memory of application, they will only execute existing code. The countermeasures discussed in this subcategory were a work-around for the fact that Intel mapped the page read permission to the page execute permission, which meant that if a page was readable, it was also executable. This has been remedied on recent versions of the Intel architecture, and NX is now available as an option in many operating systems.

**Guard page-based countermeasures** use properties of the virtual memory manager to add protection against attacks. Electric Fence [129] for example allocates each chunk of heap memory on a separate page and will place a guard page (a page without read, write or execute permissions assigned to it) behind it. If the program writes past its bounds, it writes into the guard page, which causes the program to be terminated for accessing invalid memory.

### 2.4.6 Execution monitors

Execution monitors monitor specific security relevant events (like system calls) and perform specific actions based on what is monitored. Some monitors try to limit the damage a successful attack on a vulnerability could do to the underlying system by limiting the actions a program can perform. Others detect if a program is exhibiting unexpected behavior and provide alerts if this occurs. The first type of execution monitor is called a sandbox, while the second type is called anomaly detection. In this section we also discuss fault isolation, which ensures that certain parts of software cannot cause an entire system to fail.

| Name | Type | Tech | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|---|
| Solar Designer [161] | M | DEP | S | L | Depl | Auto | OS | 0/0 |
| PaX [170] | M | DEP | All | L | Depl | Auto | OS | +/0 |
| Electric Fence [129] | D | Page | H | L | Test | Auto | Dyn | +/+ |
| Dhurjati et al. [53] | D | Page | D | M | Depl | Auto | Src | + + / − − |

Table 2.12: Paging-based countermeasures

Sandboxing is based on the "Principle of Least Privilege" [145, 146], where an application is only given as much privileges as it needs to be able to complete its task. This can be enforced in a number of ways:

- Policy enforcement: a clear policy is defined, specifying what an application specifically can and cannot do. Generally enforcement is done through a reference monitor where an application's access to a specific resource[7] is regulated. An example of such a countermeasure enforces a policy on system calls that the application is allowed to execute, making sure that the application cannot execute system calls that it would not normally need. Others attempt to do the same for file accesses by changing the program's root directory (chroot) and mirroring files under this directory structure that the program can access.

- Fault isolation ensures that certain parts of software do not cause a complete system (a program, a collection of programs, the operating system, . . . ) to fail. The most common way of providing fault isolation is by using address space separation. However, this causes expensive context switches to occur that incur a significant overhead during execution. Because the modules are in different address spaces, communication between the two modules also incurs a higher overhead. Although some fault isolation countermeasures will not completely protect a program from code injection, the proposed techniques might still be useful if applied with the limitation of what injected code could do in mind (i.e., run-time monitoring as opposed to transforming source or object code).

Many of the techniques that are used for sandboxing can be used for anomaly

[7]The term resource is used in the broadest sense: a system call, a file, a hardware device, . . .

detection. In many cases the execution of system calls is monitored and if they do not correspond to a previously gathered pattern, an anomaly is recorded. Once a threshold for anomalies is reached, the anomaly can be reported and subsequent action can be taken (e.g., the program is terminated or the system call is denied). However, attackers can perform a mimicry attack against these anomaly detectors [177] and some policy enforcers. These attacks mimic the behavior of the application that is modeled by the anomaly detector. They may be able to get the application in an unsafe state by mimicking the behavior that the detector would expect to be performed before the state is reached, but reaching the state nonetheless. For example: if an application ever performs an *execve*[8] system call in its lifetime, the attacker could execute the system calls that the detector would expect to see before executing the *execve* call.

Not all policy enforcers are vulnerable to mimicry attacks.The granularity provided by such an enforcer will influence how easy it is for an attacker to perform a mimicry attack. For example, if all system calls and their arguments are logged, mimicry becomes harder because the attackers must not only execute system calls in the correct order, they must also provide the correct arguments. If a policy enforcer is very granular, the only behavior the attacker can mimic is the behavior that was already provided by the program. An example of a granular policy enforcer is provided by Control Flow Integrity [1]. The basic policy is that the program must follow it's control flow graph: whenever the program transfers control, it must transfer it to a valid location. These locations are calculated beforehand by determining the control flow graph of the program. Then, a unique ID is assigned to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal.

| Name | Type | Tech | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|---|
| SFI [179] | C | Fault | All | L | Depl | Auto | - | $-/-$ |
| Janus [72] | C | Policy | All | M | Depl | ManL | - | $--/0$ |
| Forrest et al. [66] | D | Det | All | L | Depl | Auto | - | $+/+$ |
| MiSFIT [155] | C | Fault | All | M | Depl | Auto | - | $+/-$ |
| SASI [61] | C | Policy | All | L | Depl | ManL | - | $+/-$ |
| Naccio [63] | C | Policy | All | L | Depl | ManL | Dyn | $+/-$ |

[8]When a program calls the *execve* system call the current process is replaced with a new process (passed as an argument to *execve*) that inherits the permissions of the currently running process.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Sekar et al. [149] | D | Det | All | L | Depl | Auto | - | +/−+ |
| FMAC [134] | C | Policy | All | M | Depl | Auto | - | −−/0 |
| Wagner et al. [177] | D | Det | All | M | Depl | Auto | - | ++/−+ |
| Program Shepherding [96] | C | Policy | All | H | Depl | Auto | - | +/+ |
| Systrace [135] | C | Policy | All | M | Depl | ManS | - | −/0 |
| Yong et al. [194] | P | Policy | SHD | M | Pack | Auto | Src | +/+ |
| Linn et al. [112] | C | Policy | All | L | Depl | Auto | - | −/−− |
| Ringenburg et al. [139] | D | Policy | F | M | Pack | Auto | Src | −−/0 |
| Control Flow Integrity [1] | C | Policy | All | H | Pack | Auto | - | -+/− |
| McCamant et al. [114] | C | Fault | All | M | Depl | Auto | - | −+/− |
| Data Flow Integrity [37] | P | Policy | All | M | Pack | Auto | Src | +/+ |

Table 2.13: Execution monitors

### 2.4.7 Hardened libraries

Hardened libraries replace library functions with versions that contain extra checks. An example of these are libraries that offer safer string operations: more checks will be performed to ensure that the copy is within bounds, that the destination string is properly NULL terminated (something *strncpy* does not do if the string is too large). Other libraries check whether format strings contain '%n' in writable memory [141] (and will fail if they do) or will check to ensure that the amount of format specifiers are the same as the amount of arguments passed to the function [46].

| Name | Type | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|
| Miller et al. [118] | P | SH | L | Imp | ManL | Src | 0/0 |
| SafeStr [115] | P | SHF | L | Imp | ManL | Src | ?/− |
| Format-Guard [46] | D | F | M | Depl | Auto | Libc | − − /0 |
| Libformat [141] | D | F | L | Depl | Auto | Libc | − − /0 |

Table 2.14: Hardened Libraries

### 2.4.8 Runtime taint trackers

Taint tracking is an important type of countermeasure for web-based vulnerabilities. It is ideally suited for detecting cross-site scripting, SQL injection, command injection and other similar vulnerabilities in web applications [132, 192]. These taint trackers instrument the program to mark input as tainted[9]. If such input is used in a place where untainted data is expected (like an SQL query), an error is reported. Taint tracking can also be used to detect the vulnerabilities described in Section 2.2. In this case, the taint tracker will generate an error when an trusted memory location (like a return address) has been modified by tainted data.

| Name | Type | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|
| TaintCheck [123] | D | SHD | M | Depl | Auto | False | + + /+ |
| Chen et al. [38] | D | SHDF | M | Depl | Auto | HW + OS + False | ?/? |
| Xu et al. [192] | D | All | M | Pack | Auto | Src + False | +/? |
| Suh et al. [169] | D | SHF | M | Depl | Auto | HW + False | − − / − − |

Table 2.15: Runtime taint trackers

One important limitation with these taint trackers is that they suffer from false positives. Such a false positive can occur when a tainted data is used in a

[9]Tainted data is data which is untrusted, usually derived from input

place where untainted data is expected but is not actually vulnerable to attack. For example, if a format string is derived from tainted data and used as format specifier to *printf*, however a check has occurred to ensure that this tainted data is in fact benign. A taint tracker may report this as a vulnerabilty, while it is in fact safe code. However, most of the countermeasures described here had few to no false positives in the tests performed by the designers of the countermeasures. This can be an important limitation in situations where no false positives can be tolerated.

### 2.4.9 Dynamic analysis and Testing

Dynamic analyzers instrument the application with checks and run-time information that attempt to find buffer overflows or other vulnerabilities. Most of these tools are designed to be used during testing phases: the analysis is used instead of static analysis because more information is available at run-time. Other testing tools will generate runs of the program with specific input so that the program fails.

A number of different techniques are used to perform these checks:

**Fuzzers** generate runs of the program with specific input and monitor whether the program fails. The tools in this survey instrument the program to give information about a program to the fuzzer so that it can provide more precise input.

**General analysis** performs a similar analysis to static analyzers, but instruments the program so the program performs the checks itself and reports the results back to the user. An example of this is to add a check whether an index for an array could possibly have a value that is outside the bounds of an array.

Table 2.16 contains a number of dynamic analyzers adn testing tools.

### 2.4.10 Static Analysis

Static analyzers are generally used during the implementation or audit phases of an application, although they can also be used by compilers to decide whether a specific run-time check is necessary or not (as is done by safe languages and some other countermeasures). They do not offer any protection by themselves but are able to point out what code might be vulnerable. They operate by examining the source code of a particular application. They can be as simple as just searching the code for specific known vulnerable functions (*strcpy*, *gets*, . . . ), or as complicated as building a complete model of the running application.

However, determining statically, for any possible input program, whether a program will contain an overflow or not, is undecidable: it is trivial to reduce this problem to the halting problem [136]. So all analyzers will contain a number of

| Name | Type | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|
| STOBO [78] | P | SH | M | Test | ManS | Src | ?/? |
| Purify [77] | P | HD | M | Test | Auto | - | $++/+$ |
| Muse [104] | P | SHDI | M | Test | Auto | Src | $++/-+$ |
| FIST [70] | P | S | M | Test | ManS | Src | ?/? |
| Fink et al. [65] | P | All | M | Test | ManL | Src | ?/? |
| OBOE [33] | P | S | L | Test | Auto | - | $+/-$ |
| Cadar et al. [36] | P | SH | M | Test | ManS | Src | ?/? |

Table 2.16: Dynamic Analysis and Testing

false positives (the amount of correct code that is incorrectly reported as being vulnerable), false negatives (vulnerable code that is not reported as such) or both. The main criteria for determining the effectiveness of a source code analyzer is the false positive to total warnings ratio, the completeness (i.e., are false negatives possible or not) and how well they scale to larger, real-world software systems. It is important to note the difference between sound and unsound static analyzers: sound analyzers will find all possible overflows, usually at the expense of generating more false positives and being less scalable while unsound analyzers try to find the right balance between false positives, false negatives and being able to analyze larger systems than sound analyzers.

Several types of analyzers exist. In Table 2.17 we have grouped the analyzers into subtypes based on the main analysis that is performed. Since all analyzers are used at the implementation stage, we have omitted the stage column from the table for these countermeasures.

- Lexical analyzers are very simple and fast. Unlike most static analyzers they do not parse the source code to do analysis, instead they analyze a program at the lexical level and match code to a vulnerability database that contains a list of unsafe functions or functions that are misused in many cases. They then display a potential risk factor and a description of the possible vulnerability. When they have identified possibly vulnerable functions they match them against possible safe or unsafe uses of the functions and determine if they are to be reported and with which level of severity. For example $strcpy(dst, "\backslash n")$; will usually not be reported as a potential vulnerability because the source string is fixed and consequently the potential risk factor is extremely low.
- Verifiers convert the program into a language that lends itself better to verification. Using rules that specify preconditions, side effects and postconditions

of different operations, the program is augmented with this information. The verifier will then ensure that these pre- and postconditions are adhered to.

- Taint analyzers mark all data derived from an untrusted source (e.g., the network, user input, etc.) as tainted. If they detect that this data is later used in a place where untainted data is expected (e.g., as a format specifier for a format string function) without first being marked as safe, they generate an error.
- Symbolic executors are static analyzers that build an execution model of the source code they are analyzing. They simulate execution of actions performed by execution paths and use this to build a model of the program. The results of these actions are then assigned to the values associated with them in the program. When such a value is later used, the analyzer can more accurately determine whether the usage is safe.
- Integer range analyzers model arrays in programs as a number of integers, usually denoting the total number of bytes allocated to an array and the number of bytes in use by the array when an action is taken. These analyzers generate constraints that must hold after a specific operation is performed on an array. If a constraint can be violated by an operation, it is reported as a possible overflow.

Table 2.17: Static Analyzers

| Name | Type | Tech | Vulns | Assur. | Stage | Effort | Lim | Cost |
|---|---|---|---|---|---|---|---|---|
| Splint [103] | P | Avrfy | SHF | M | Imp | ManL | Src | −/− |
| CSSV [57] | P | Avrfy | SH | H | Imp | ManL | Src | +/ + + |
| Shankar et al. [152] | P | Ataint | F | M | Imp | ManL | Src | −/− |
| Meta-compil-ation [6] | P | Ataint | I | M | Imp | ManL | Src | ?/? |
| PREfix [35] | P | Asym | SHD | M | Imp | ManS | Src | ++/+ |
| PREfast [105] | P | Asym | SHD | M | Imp | ManS | Src | −/− |
| ARCHER [189] | P | Asym | SHD | M | Imp | ManS | Src | +/? |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Simon et al. [154] | P | Avrfy | SH | H | Imp | ManS | Src | ?/? |
| BOON [178] | P | Aint | SH | M | Imp | ManS | Src | -+/? |
| Rugina et al. [143] | P | Asym | SH | M | Imp | ManS | Src | ?/? |
| Ganapathy et al. [69] | P | Aint | SH | M | Imp | ManS | Src | ?/? |
| Eau Claire [40] | P | Avrfy | SH | M | Imp | ManS | Src | ?/? |
| ITS4 [174] | P | Alex | SHF | L | Imp | ManL | Src | −−/−− |
| Flaw-Finder [181] | P | Alex | SHF | L | Imp | ManL | Src | −−/−− |
| RATS [148] | P | Alex | SHF | L | Imp | ManL | Src | −−/−− |
| Hackett et al. [76] | P | Avrfy | SH | M | Imp | ManS | Src | ?/? |
| Blanchet et al. [25] | P | Asym | SHD | M | Imp | ManL | Src + Chg | +/+ |
| UNO [81] | P | Avrfy | SH | M | Imp | ManS | Src | −/− |

## 2.5 Related work

A lot of work has been done in categorizing vulnerabilities [2, 7, 21, 22, 23, 24, 82, 102] in many different taxonomies. Each of these has its own relative strengths and weaknesses, However, little work has been done in categorizing the available countermeasures to some important vulnerabilities. Cowan et al. [50] have examined some countermeasures to buffer overflows. They examine a set of defenses including bounds checking modifications (Compaq c-compiler, Jones&Kelly compiler) safe languages (Java and ML), debugging tools (Purify), library patches (Snarskii FreeBSD patch) and compiler modifications (StackGuard and an early idea for PointGuard). Most of these countermeasures are only mentioned briefly, while only StackGuard and PointGuard are examined more in depth.

Wilander and Kamkar have published two documents in this area, one relating to static countermeasures [186] and one describing dynamic ones [187]. Wilander and Kamkar's documents are limited to publicly available tools with most focus being on comparing existing production tools. In [186], they examine ITS4, Flawfinder, RATS, Splint and BOON for comparison while in [187] they compare

StackGuard, Stack Shield, Propolice, Libsafe and Libverify. In these documents they examine how effective these countermeasures are at respectively finding vulnerabilities in source code and preventing a wide range of attacks.

Lhee and Chapin [111] performed an extensive survey on techniques for exploiting buffer overflows and format string vulnerabilities. While they focus mainly on exploitation techniques, they also provide an overview of some countermeasures designed to stop buffer overflows and format string vulnerabilities. One of our main goals is to provide a comprehensive survey of countermeasures for vulnerabilities that could allow code injection. To achieve this, our vulnerability section contains more vulnerabilities: dangling pointer references and integer errors are not considered in the Lhee and Chapin paper. Because of the difference in focus we also discuss more countermeasures and provide a synthesis that allows the reader to weigh the advantages and disadvantages of using one specific countermeasure as opposed to using another more easily.

Erlingsson [60] performed a survey in parallel to our survey which examines a number of attacks against and defenses for C-like languages.

## 2.6 Conclusion

Stack-based buffer overflows have been a very important source of vulnerabilities in programs written in C. It was the type of attack most often found and was the first type of vulnerability that was exploited to gain code execution. Since then many countermeasures have been designed that protect against attacks against stack-based buffer overflows. However attackers have also discovered new techniques to exploit vulnerabilities in C programs: besides stack-based buffer overflows, other types of vulnerabilities were soon discovered. Attackers also designed several approaches to bypass countermeasures. This has led to more complex countermeasures that attempted to solve these problems. Which in turn led to a classic arms race between attackers and countermeasure designers. A classic example of this is StackGuard [49]. Due to its popularity attackers started looking for new ways to exploit stack-based buffer overflows that would bypass the protection offered by this type of countermeasure [34]. This led to the development of Propolice [62], which countered the technique used to bypass StackGuard described in [34]. Further countermeasures addressed some of the shortcomings of these countermeasures, which could still allow an attacker to bypass the countermeasure.

Attackers also discovered new types of vulnerabilities and as a result, new countermeasures were designed to protect against these vulnerabilities and new methods of bypassing them were discovered.

In this survey we presented an overview of the most commonly exploited vulnerabilities that lead to code injection attacks. More importantly however, we also presented a survey of the many countermeasures that exist to protect against these vulnerabilities together with a framework for evaluating and classifying them. We described the many techniques used to build countermeasures and discussed some of their advantages and disadvantages. We also assigned specific properties to each of the countermeasures that allow the reader to evaluate which countermeasures could be most useful in a given context.

Although this survey tried to be as complete as possible when discussing the different countermeasures that exist, it can never be entirely complete. Countermeasure design is an active field, so this survey can only provide a snapshot of the current state of the field with respect to specific countermeasures. However, we believe we have a strong framework that can be applied to future countermeasures to further evaluate and classify these new countermeasures.

# Chapter 3

# Extended protection against stack smashing attacks without performance loss

*This chapter contains a paper that discusses a countermeasure that provides efficient protection against stack-based buffer overflows. Based on what was learned from conducting the survey in Chapter 2, a methodology [199] was designed to more effectively design countermeasures against code injection attacks. During the design of this methodology we suggested the idea of separating control-flow data (data that is used to regulate the control flow of the program, like a return address) from regular data, similarly to how many operating systems separate code from data. When this was applied to the stack, a number of other types of data became relevant that could also be used by attackers to gain code execution (e.g., by modifying data pointers to perform indirect pointer overwriting or by modifying an integer which is later used as an offset in pointer arithmetic). An evaluation of all the types of data that are stored on the stack was performed and based on this evaluation, each data type was assigned a target and source value. The target value determines how useful the memory location is to attackers if they were to control it. The source value determines how useful this type of data would be to an attacker when performing an actual attack. Based on the values assigned during the evaluation of the data types, the stack is then split into multiple stacks, preventing data with different source and target values from overwriting each other. This separation of code data from regular data provides efficient protection against stack-based buffer overflows without relying on random num-*

*bers that must remain secret, offering protection even in the face of an attacker who has complete read access to the address space of the process.*

*This paper was written together with Davide Pozza, Frank Piessens and Wouter Joosen. It was published in the proceedings of the Twenty-Second Annual Computer Security Applications Conference [201].*

## Abstract

In this paper we present an efficient countermeasure against stack smashing attacks. Our countermeasure does not rely on secret values (such as canaries) and protects against attacks that are not addressed by state-of-the-art countermeasures. Our technique splits the standard stack into multiple stacks. The allocation of data types to one of the stacks is based on the chances that a specific data element is either a target of attacks and/or an attack source. We have implemented our solution in a C-compiler for Linux. The evaluation shows that the overhead of using our countermeasure is negligible.

## 3.1 Introduction

Buffer overflow vulnerabilities are a significant threat to the security of a system. Most of the existing buffer overflow vulnerabilities are located on the stack, and the most common way for attackers to exploit such a buffer overflow is to use it to modify the return address of a function. By making the return address point to code they injected into the program's memory as data, they can force the program to execute any instructions with the privilege level of the program being attacked [3].

According to the NIST's National Vulnerability Database [120], 584 buffer overflow vulnerabilities were reported in 2005, making up 12% of the 4852 vulnerabilities reported that year. In 2004 the amount of reported buffer overflow vulnerabilities was 341 (14% of 2352). This means that, while the amount of reported vulnerabilities has almost doubled in the past year, buffer overflows still remain an important source of attack. 418 of the 584 buffer overflows reported in 2005 had a high severity rating, this makes up 21% of the 1923 vulnerabilities rated with a high severity level. They also make up 42% of the vulnerabilities that allow an attacker to gain administrator access to a system.

Stack-based buffer overflows have traditionally made up the largest bulk of these buffer overflows, and are the ones most easily exploited by attackers. Many countermeasures have been devised that try to prevent code injection attacks [198]. Several approaches attempt to solve the vulnerabilities entirely [8, 89, 94, 124, 144, 193]. However, they generally suffer from a substantial performance impact. Other types of countermeasures have been developed with better performance results that specifically target stack-based buffer overflows. These countermeasures can be divided into four categories. A first category [49, 62] offers protection by using a random value, which must be kept secret from an attacker. If the program leaks this information (e.g., through a 'buffer over-read' or a format string vulnerability) the protection can be bypassed entirely. A second category [11, 42, 172, 191] copies the return address and the saved frame pointer, and compares or replaces them when the function returns. While this protects against the return address being overwritten, it does not protect other information stored on the stack (e.g., pointers) that could be used by an attacker to execute arbitrary code. A third category tries to correct the library functions that are typically the source of an overflow (e.g., *strcpy*) [11]. however, these libraries do not protect against buffer overflows that could occur at a different place in the program (e.g., an overflow caused by a loop). A fourth category tries to make attacks harder by modifying the operating system [12, 18, 93, 161, 170] or hardware [109, 191].

In this paper we present a new approach for protecting against stack based buffer overflows by separating the stack into multiple stacks. This separation is done according to the type of data stored on the stack. Each stack is protected from writing into the other stack by a guard page[1]. Our countermeasure offers equal or better performance results than the countermeasures in the categories discussed earlier and does not suffer from some of their weaknesses: it does not rely on random numbers and protects pointers as well as the return address and frame pointer. In [199] we describe a more global approach to separating control flow data from regular data and in [200] we discuss applying it to the heap to separate the metadata from the regular data.

The paper is structured as follows: Section 3.2 briefly describes the technical details of the stack-based buffer overflow, some representative countermeasures and their weaknesses. Section 3.3 discusses the design and implementation of our countermeasure. Section 3.4 evaluates our countermeasure in terms of performance and security. In Section 3.5 we discuss limitations of and possible improvements to our approach and describe ongoing work. Section 3.6 compares our approach to existing countermeasures, while Section 3.7 presents our conclusions.

## 3.2 Stack-based buffer overflows[2]

Buffer overflows are the result of an out of bounds write operation on an array. In this section we briefly recap how an attacker could exploit such a buffer overflow on an array that is allocated on the stack.

When an array is declared in C, space is reserved for it and the array is manipulated by means of a pointer to the first byte. At run-time no information about the array size is available and most C-compilers will generate code that will allow a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in such adjacent memory space, it could be possible for an attacker to overwrite it. On the stack this is usually the case: it stores the addresses to resume execution at after a function call has completed its execution, i.e., the return address.

[1]A guard page is page of memory where no permission to read or to write has been set. Any access to such a page causes the program to terminate.

[2]This section discusses the problem of stack-based buffer overflows, a more extensive discussion on the topic of vulnerabilities can be found in section 2.2

For example, on the IA32-architecture the stack grows down (i.e., newer function call have their variables stored at lower address than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments to a function that was called, registers whose values must be stored across function calls, local variables, the saved frame pointer and the return address. An array allocated on the stack will usually be contained in the section of local variables of a stackframe. If a program copies data past the end of this array it will overwrite anything else stored before it and thus will overwrite other data stored on the stack, like the return address.

Several countermeasures were designed against this attack: ranging from bounds checkers to operating system changes. Many of these are discussed in Section 3.6. Here, we discuss two of the most used countermeasures that protect against this attack.

StackGuard [49] was designed to be an efficient protection against this type of attack: it protects the return address by placing a randomly generated value (called a canary) between the saved frame pointer and the local variables on the stack. This canary is generated at program start up and is stored in a global variable. When a function is called, the countermeasure puts a copy of the canary onto the stack after the saved frame pointer. Before the function returns, the canary stored on the stack is compared to the global variable: if they differ, the program will be terminated. If an attacker would want to overwrite the return address, he would have to know the canary, so he could replace it. A significant problem with this approach is the fact that the program cannot leak the canary, if it did, the attacker could just write the correct value back on the stack and the protection would be bypassed.

Figure 3.1 depicts the stack layout of a program protected with StackGuard and illustrates an attack called indirect pointer overwriting [34]. This attack consists of exploiting a local buffer to overwrite a pointer *p1* stored in the same stackframe and to make the pointer refer to the return address. When the pointer is later dereferenced for writing, it overwrites the return address rather than the value it was originally pointing to. If attackers can control the value that the program writes via the pointer, they can modify the return address to point to their injected code.

ProPolice [62] attempts to protect against this type of attack by reorganizing the local variables stored in each stackframe: all arrays are stored before all other local variables in each stackframe. This prevents an attacker from using an overflow to overwrite a pointer and using an indirect pointer overwrite to bypass the protection.

However, as mentioned earlier, this type of protection has some limitations: if a program leaks the canary (e.g., through a format string vulnerability or a 'buffer over-read'), the protection can be bypassed completely. Another point of attack

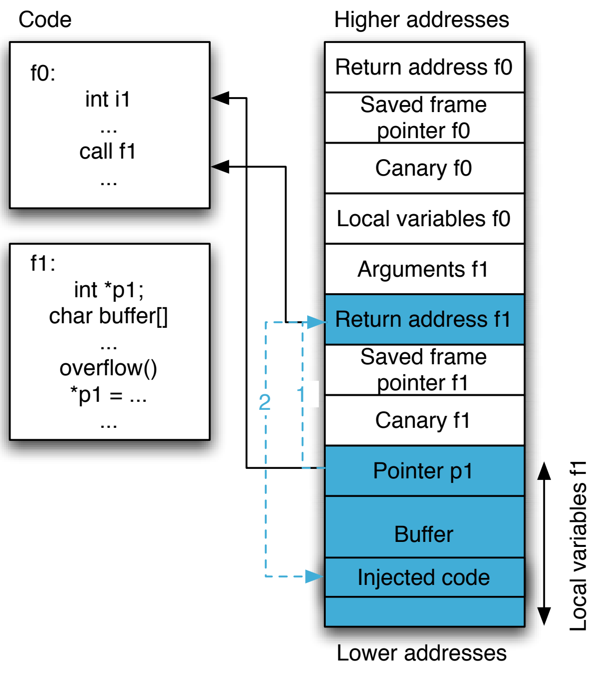


Figure 3.1: Indirect pointer overwriting attack

is to use a buffer overflow to overwrite an array of pointers[3] in a program, or to use a structure that contains a buffer to overwrite another structure that does contain a pointer. It also does not protect against memory that is allocated with the *alloca*[4] function, if an overflow occurs in memory allocated using this call, it could be used to perform an indirect pointer overwrite.

In the next section we discuss our approach that aims to better protect against these attacks, while still preserving or improving the performance of the previously described countermeasures.

## 3.3 The multiple stacks countermeasure to protect against buffer overflow vulnerabilities

This section describes the approach of the multiple stacks countermeasure by describing the basic concepts behind its design, as well as how it was implemented.

### 3.3.1 Approach

The stack stores several kinds of data: some data is related to control flow, such as stored registers, but it also contains regular data like the local variables of a function. If the control data is modified, an attacker may be able to inject code. However, modifying the regular data can sometimes also be used by an attacker to inject code (e.g., pointers could allow indirect pointer overwriting).

In this section we describe an approach which separates the stack into multiple stacks based on two criteria: how valuable data is to an attacker when it is a target for attack, and the risk of the data being used as an attack source (i.e., abused to perform an attack). These properties are not mutually exclusive: some data could be both a target and a source. So, we must evaluate all possible data types and place them in categories according the risk of being an attack source and the effective value.

We can assign data a ranking based on its risk of being an attack source and the value it has as a target. Data can have a low, medium, or high ranking for both properties (e.g., the return address has a high target value because attackers generally want to overwrite it, and a low source value because it can't be attacked directly). Based on these rankings we can divide the data into different categories. This is illustrated in Table 3.1, where we use six categories.

In principle, one can always argue for other categories or combinations. However, we decided to limit these categories to six based on how we perceive the

[3]an array of pointers is a contiguous memory region containing only pointers, no structures

[4]*alloca* is used to dynamically allocate space on the stack, it behaves in much the same way as the *malloc* function, except that the memory it allocates will be released when the function returns.

Table 3.1: Attack source versus attack target categories

| Source \ Target | Low | Medium | High |
|---|---|---|---|
| Low | cat. 3 | cat. 2 | cat. 1 |
| Medium | cat. 5 | cat. 3 | cat. 2 |
| High | cat. 5 | cat. 4 | cat. 6 |

combined risk/value resulting from the combination of attack source risk and attack target value. We believe that the presented set of six categories is a good approximation. The main objective of this section is to show that a multiple stacks countermeasure (based on several categories) can be supported efficiently.

**Category one** contains highly valuable data and there is only a low risk of it being used as an attack source. This is the main category that we wish to protect from buffer overflows.

**Category two** represents two cells from the summary table: data which has a low risk of being an attack source, but a medium target value, and data that has a medium risk of being a source, but is also a high-value target. We consider both these two types of data to have a similar combined risk/value.

**Category three** contains data which has a medium risk of being a source, but is also only a medium-value target. We have supplemented it with data that has the least importance in our countermeasure: low on source-risk and low on target-value. Mainly, it does not matter where this type of information is placed since it needs no protection and can't be used to attack. As such, we decided on placing it in a middle category.

**Category four** contains data that has a high risk of being an attack source, but which is also a medium-value target. So, there is some need for protection.

**Category five** contains data that has a high or medium risk of being a source, but has only low value as a target. It contains both high and medium risk data, because the data needs to be isolated from higher-value targets, but does not need to be protected.

**Category six** is the hardest data to protect. It is both a high-value target and has a high risk of being used as an attack source. We place it in a separate category because it needs both extra protection and we need to protect other data from this type of data.

We can now decide what information to put in each of these categories by assigning them rankings of their target-value and attack source-risk.

**The Return address** is the most obvious target for attack: if an attacker can modify it, he can easily execute injected code. However, an attacker does not directly control the return address, so it is an unlikely source.

- Attack target: High; Attack source: Low

**Other saved registers** on the stack, like the saved frame pointer and the caller-save and callee-save registers could be used to attack a program [97]. So, all these are valuable targets, but generally an attacker cannot use them to mount an attack.

- Attack target: High; Attack source: Low

**Pointers** can contain reference functions or data. If a function pointer is overwritten, an attacker can directly execute injected code. If a data pointer is overwritten, an attack could use indirect pointer overwriting, so these are very likely targets for attacks. However, they cannot be used as an attack source, unless they can be modified by an attacker.

- Attack target: High; Attack source: Low

**Integers** can sometimes be used to store pointers or indices to pointer operations, so they can be considered attack targets. They are not attack sources in the sense that they could directly overwrite other information on the stack.

- Attack target: Medium; Attack source: Low

**Floating types** are not valuable targets because they will not generally contain information that could lead to code injection (either directly or indirectly). They are also unlikely attack sources because they can't be used directly to overwrite adjacent memory locations.

- Attack target: Low; Attack source: Low

**Arrays** are assigned different target values and attack source-risks depending on their type:

**Arrays of pointers** are valuable targets, because they contain pointers, and as such could be used to perform an indirect pointer overwrite, if modified. However, there is also a chance that an operation on an array of pointers could lead to writing outside the bounds of the array. Thus, there is a risk of it being used as an attack source as well. However, these type of arrays are not generally used with functions that are prone to buffer overflows (e.g., *strcpy* and related functions), so this risk is not as high as with arrays of characters.

- Attack target: High; Attack source: Medium

**Arrays of characters** are the traditional arrays that are most vulnerable to buffer overflows. The risk of them being used as an attack source is high, especially since they are also often used with unsafe copying functions. They do not contain any information that could indirectly or directly lead to a code injection attack.

- Attack target: Low; Attack source: High

**Other arrays** are possible targets because an integer in an array of integers could be used to store a pointer. As with arrays of pointers, they are possible sources, since an out of bounds write could occur, but they are not generally used with the most dangerous functions.

- Attack target: Medium; Attack source: Medium

Arrays of structures and unions are discussed separately.

**Structures/unions** are assigned different target values and attack source-risks depending on the type of the data they contain:

**Structures containing no arrays** at any level (structures and unions can contain other structures or unions) are unlikely attack sources, but possible targets because they possibly contain pointers.

- Attack target: Medium; Attack source: Low

**Structures containing arrays of characters** are likely sources because a buffer overflow could occur in the character array. They are also possible targets because they could contain pointers.

- Attack target: Medium; Attack source: High

**Structures containing other arrays** are possible sources, because overflows could occur. They are also a target because the structure or union could be used to store a pointer.

- Attack target: Medium; Attack source: Medium

**Arrays of structures/unions** Arrays of structures are a special case because the structures or unions stored in such an array can contain arrays at some level.

**Not containing arrays of characters** If the structures or unions inside the array do not contain arrays of characters at any level, we treat them the same as other arrays: possible targets and possible sources.

- Attack target: Medium; Attack source: Medium

**Containing arrays of characters** As previously mentioned for structures or unions containing character arrays: they are a likely source, and a possible target.

- Attack target: Medium; Attack source: High

Based on these assignments and Table 3.1, the different categories contain the following data:

**Category one** : return address, other saved registers, pointers.

**Category two** : arrays of pointers, structures and unions (no arrays), integers.

**Category three** : floating types, other arrays, structures/unions containing arrays but not arrays of characters at any levels, arrays of structures that do not contain arrays of characters at any level.

**Category four** : structures containing array of characters, arrays of structures containing arrays of characters

**Category five** : arrays of characters

Category six is the hardest to protect, thankfully it is empty in our risk/value evaluation. There is no data on the stack that we consider to have high risk of being an attack source but is also a high-value target. An exception to this is in a program where an array of characters is passed to an interpreter (e.g., the system() function). This data could be stored in category six. However, since the existence of this type of data is very application-specific we consider it outside the scope of our countermeasure.

As with the different categories, the actual value that we have assigned specific data is based on the value or risk that we perceive it to have. If some data would be assigned a different risk or value by someone else, resulting in it being placed in a different category, this would only require minimal modification of our existing countermeasure.

The main principle used to design this countermeasure is to separate information in these different categories from each other by storing them on separate stacks. As such they can no longer be overwritten by information that has been moved to a different stack. Figure 3.2 depicts the memory layout if we were to map the five categories that contain data onto five different stacks.

It is however fairly simple to modify our design (and our implementation) to support other stack configurations depending on the amount of risk that these data types or categories present (or if the risk of a particular category or data type can be diminished or abolished entirely) in a particular application versus the amount of memory that can be used. An example of this would be to support only two stacks, and to place categories one, two and three on the first stack, while storing categories four and five on the second stack.

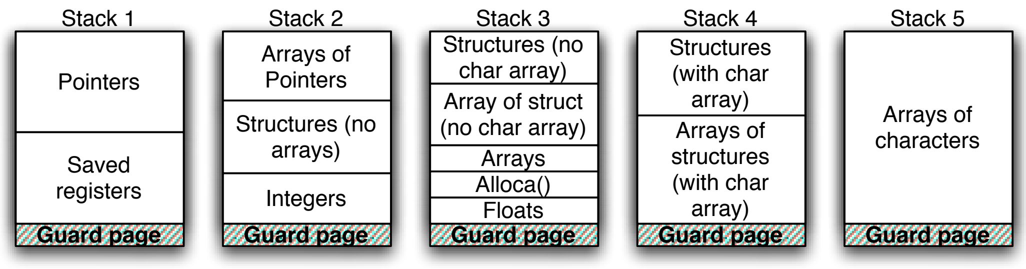


Figure 3.2: Stack layout for 5 stacks

### 3.3.2 Implementation

The multiple stack countermeasure was implemented in gcc-4.1-20050902 for Linux on the IA32 architecture. Each stack is stored sequentially after the other and each stack is protected from the previous one using a guard page. We start off by allocating the different stacks at a fixed location from one another. This fixed location is the maximum size that the stack can grow to (this must be known at compile time). As long as no information is written to the specific pages that were allocated for the stack, the program only uses virtual address space, rather than physical address space so we can easily map all stacks into memory without wasting any physical memory[5].

The countermeasure was implemented in the pass of the compiler that converts the GIMPLE representation[6] into RTL[7].

We implement our countermeasure by modifying the way local variables are accessed in a function. When a function is called in a program, the return address is stored on the stack. To access local variables of a function, the current value of the register containing the stack pointer is copied to the frame pointer register (and the current frame pointer is saved on the stack). This frame pointer is used as a fixed location to access a function's local variables (all variables are accessed as an offset to the frame pointer), this mechanism is used because the value of the register containing the stack pointer is changes whenever a variable is pushed or popped from the stack. The compiler calculates the offset to the frame pointer for local variables at compile time and uses this offset whenever it accesses this variable. When the function returns, the saved frame pointer is restored into the

[5]While this will not cause physical memory overhead, the use of a large amount of virtual address space will use up more page table entries, resulting in performance overhead.

[6]GIMPLE is a language- and target-independent tree representation of the program being compiled. The compiler will convert the program into static single assignment form (SSA) at this level.

[7]RTL is the register transfer language, a language-independent, but target-dependent, intermediate representation used by the compiler to do some optimizations.

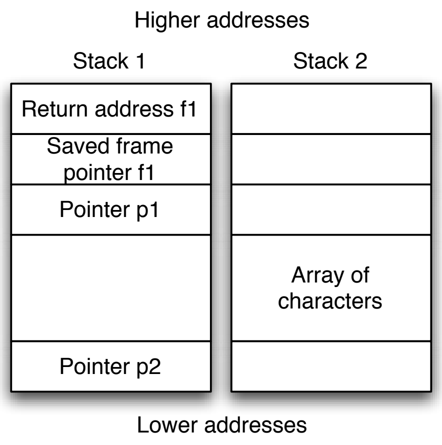


Figure 3.3: Gaps on the different stacks

frame pointer register.

We use this mechanism to efficiently implement our countermeasure: instead of using multiple stack pointers, we modify the offset to the frame pointer that is used to access the variable. We add $(stacknr - 1) * (sizeofstack + pagesize)$ to the offset, which results in the access of the variable on the correct stack. As a consequence, all operations that use this variable use the correct stack to address it. This also means our countermeasure doesn't incur any overhead because the offset is simply a larger constant value, but the instruction to access it remains the same[8].

Because the program is instrumented in this way, the stack pointer remains unchanged and effectively controls all five stacks. The advantage is that *setjmp* and *longjmp*[9] work unchanged. The drawback of this countermeasure is that it results in gaps on the remaining stacks, resulting in wasted memory. Figure 3.3 depicts this for two stacks. We provide a more detailed discussion on the memory overhead in Section 3.4.

A special case, that we did not address in the design and the categories above, is memory allocated with *alloca*. The information stored in it could be both an attack source to overwrite other memory and could contain information that could be used to perform a code injection attack (e.g., a function or data pointer). The third category contains data which has both a medium risk of being a source of

[8]This is true for the IA32 architecture, but architectures which have a maximum offset size may incur a higher overhead because extra instructions are needed to calculate the offset

[9]The *longjmp* function jumps to the most recent place in the code where a *setjmp* was executed, resetting the stack pointer (and other registers) to the value they held at the moment *setjmp* was called.

attack, but also contains data which has a medium target value. Since this is exactly what the memory allocated by *alloca* would fall under, it is placed on stack three. Given this, we chose to modify this call to allocate memory on stack three in the case of five stacks and stack two in the case of two stacks.

## 3.4 Evaluation

To test the performance overhead, we ran several benchmarks on programs instrumented with our countermeasure (running with 5 stacks) and without. All tests were run on a single machine (Pentium 4 2.80 Ghz, 512MB RAM, no hyperthreading, running Ubuntu Linux 5.10 with kernel 2.6.12.10). The GCC compiler version 4.1-20050902 was used to compile all benchmarks.

### 3.4.1 Performance

This section evaluates our countermeasure in terms of performance overhead. Both macro- and microbenchmarks were performed.

#### Macrobenchmarks

We ran the full SPEC® CPU2000 Integer reportable benchmark [79], which gives us an idea of the overhead associated with general-purpose single-threaded programs. All programs in benchmarks except for 252.eon (it is written in C++, while our prototype implementation is only for C) were used to perform these benchmarks.

Table 3.2 contains the amount of code present in a particular program (expressed in lines of code), the runtime in seconds[10] when compiled with the unmodified gcc and the runtime when compiled with our multistack countermeasure. The results in this table show that the performance overhead of using our countermeasure is negligible for most of these programs. There is a slightly higher overhead of 2-3% for the programs *vortex* and *twolf.* The negative overheads in the table are so low that they can be attributed to normal variations between runs and, as such, these overheads can be considered equivalent.

To determine which stacks were used most, we statically determined how many local variables were stored on the stack for each program. The numbers in Table 3.3 reflect the local variables which are assigned stack positions by the compiler and are then separated onto different stacks by our countermeasure. If a program has 0 local variables for a specific stack, that does not mean that the program does not make use of data types that may be stored on this stack: the compiler may decide to put a local variable solely in registers, which would mean it wouldn't be part of

[10]Since the results in this table represent one run of the benchmarks, no standard error is specified

| **SPEC CPU2000 Integer benchmarks** | | | | |
|---|---|---|---|---|
| Program | LOC | Gcc 4.1 (s) | Multistack (s) | Overhead |
| 164.gzip | 8,616 | 201 | 201 | 0% |
| 175.vpr | 17,729 | 213 | 212 | -0.47% |
| 176.gcc | 222,182 | 89.7 | 89.8 | 0.11% |
| 181.mcf | 2,423 | 248 | 249 | 0.4% |
| 186.crafty | 21,150 | 116 | 115 | -0.86% |
| 197.parser | 11,391 | 257 | 255 | -0.78% |
| 253.perlbmk | 85,185 | 150 | 151 | 0.67% |
| 254.gap | 71,430 | 101 | 101 | 0% |
| 255.vortex | 67,220 | 169 | 174 | 2.96% |
| 256.bzip2 | 4,649 | 204 | 203 | -0.49% |
| 300.twolf | 20,459 | 291 | 297 | 2.06% |
| **Microbenchmarks** | | | | |
| loop | 20 | $9.166 \pm 0.029$ | $9.2 \pm 0.015$ | 0.37% |
| fibonacci | 14 | $3.354 \pm 0.004$ | $3.363 \pm 0.005$ | 0.27% |

Table 3.2: Benchmark results of the multistack approach

| **SPEC CPU2000 Integer benchmarks** | | | | | |
|---|---|---|---|---|---|
| Program | Stack 1 | Stack 2 | Stack 3 | Stack 4 | Stack 5 |
| 164.gzip | 4 | 8 | 9 | 0 | 3 |
| 175.vpr | 9 | 36 | 17 | 0 | 21 |
| 176.gcc | 150 | 217 | 171 | 2 | 109 |
| 181.mcf | 1 | 6 | 0 | 0 | 1 |
| 186.crafty | 0 | 15 | 17 | 3 | 30 |
| 197.parser | 0 | 11 | 2 | 0 | 11 |
| 253.perlbmk | 43 | 349 | 18 | 0 | 37 |
| 254.gap | 18 | 7 | 1 | 0 | 25 |
| 255.vortex | 664 | 1092 | 7 | 10 | 27 |
| 256.bzip2 | 0 | 0 | 11 | 0 | 8 |
| 300.twolf | 0 | 79 | 11 | 0 | 26 |
| Total | 889 | 1820 | 264 | 15 | 298 |

Table 3.3: Local variables stored on each stack

our measurements. Besides local variables like pointers, stack 1 will always contain the return address and other saved registers which are automatically saved to this stack by the compiler. Since these must not be separated by our countermeasure, they are also not part of the measurement described here.

#### Microbenchmarks

Two programs that make extensive use of the stack were run as a microbenchmark. One program (called loop in the table) which simply calls a function 1 million times. This function performs an addition of two local variables (filled with 'random'[11] values), fills a local array with this random value, and allocates and frees a chunk of random size. The second program performs a recursive Fibonacci calculation of the 42nd Fibonacci number. These programs were each run 100 times both compiled with the unmodified gcc and our multistack countermeasure. Table 3.2 contains the average runtime in seconds, followed by the standard error for both versions (since the SPEC CPU2000 benchmark was run a single time, no standard error is reported for it). Both the unmodified gcc and our multistack countermeasure have very comparable performance, we conclude from this that our modifications only add negligible overheads for function entry and exit. The main performance overhead found in the macrobenchmarks is likely due to caching issues.

These results also confirm that the performance overhead of using our countermeasure is negligible.

### 3.4.2 Memory overhead

The maximum memory overhead of this countermeasure will be the original stack usage multiplied by the number of stacks that are used.

Because variables are accessed by simply adding a constant value to the frame pointer, we end up with gaps on all stacks and waste space on all stacks. To reduce the waste, it is possible to implement a version where we calculate the actual location that the variable is on for every stack. This would eliminate gaps in a function entirely. Some gaps would still exist between function calls (because we still only have one stack pointer), but these could be reduced to be equal to the amount of space used on the largest stack. This still allows us to use a single stack pointer, because all other stacks will continue to have gaps, but these gaps will be smaller than in the current implementation. Since all these calculations can be done at compile time, no extra performance overhead would be incurred.

[11] We use a fixed seed for the random function, so the generated values are the same over different runs.

## 3.5 Discussion and ongoing work

Most applications will never increase the default stack size. However, applications that do need a larger stack, may be limited in the size their stack may grow to a predetermined maximum, since the location of the stacks must be set to a fixed location when the program is compiled. If the maximum size that the stack could grow to is known beforehand (programs that need more stack space will increase their stack size with a system call at runtime, however often this new size is provided as a constant in the source code, rather than dynamically calculated), the locations of the different stacks can easily be changed to accommodate a larger stack. The application would only lose virtual address space when moving the stacks further apart and would not use any extra physical memory until the data is written to these pages. We discuss a possible solution to this problem below.

Multithreaded programs which use many threads, may suffer from high performance overheads when implementing this countermeasure: the amount of memory and virtual address space that would be needed to set up multiple stacks for each thread may be prohibitively expensive.

Our approach is incompatible with most address space layout randomization (ASLR) [170] implementations. This can be mitigated by finding the start of the stack dynamically when the program is started, while setting up the extra stacks. This can be done either by recursively following the saved frame pointer values or by modifying the ASLR implementation to store the value in a known location (e.g., the normal stack location) and subsequently clearing it when the multiple stacks have been set up.

Because not all applications can afford to use five stacks, but would still like more security than simply reducing the amount of stacks to two can offer, the multistack countermeasure could be extended by a concept that we call *selective bounds checking*. *Selective bounds checking* only bounds checks write operations to some types of arrays to prevent them from being overflowed. If, for example, we can bounds check write accesses to arrays of pointers, we could determine that the risk of the array of pointers being used as an attack source is reduced to a low enough level so that we can place it in the first category. While the bounds checking for direct access is straightforward (the program is instrumented to dynamically check if the index is within the bounds of the array), static analysis is needed to determine how to instrument indirect accesses to an array. This means that this bounds checker will not find all cases of such accesses, but since we are only interested in reducing the risk of already unlikely attack sources (like arrays of pointers). Because most programs do not operate heavily on these unlikely sources, the performance overhead of adding this type of bounds checking may be acceptable.

This selective bounds checker can be applied to reduce the number of stacks to two, which could make it realistic to reserve a register as a stack pointer for the second stack. This would allow us to place this second stack anywhere in memory, which would solve the fixed stack size problem, the incompatibility with ASLR and would eliminate the gaps. However, a performance overhead will probably be incurred because this extra register must be modified in much the same way as the original stack pointer.

One vulnerability that is present in existing countermeasures, that we did not address in our countermeasure either is the fact that a structure can contain both a pointer and an array of characters, giving the attacker the possibility to overwrite this pointer using the array of characters. The same is true for memory allocated with alloca (it can be used to store array of characters and pointers). This is an important limitation that a lot of countermeasures suffer from.

A non-control data attack [39] that relies on modifying a character array would still work, but is severely limited to only being able to overwrite character arrays.

Our approach also does not *detect* when a buffer overflow has occurred. However, it is possible to easily and efficiently add such detection as an extension to our implementation by using the technique used by StackGuard and Propolice of placing a random number on the stack and verifying it before returning from the function. This canary would be placed on every stack and compared to the value stored on the first stack before returning. Since the random number is mirrored, we can also use a per function canary, rather than a global one, reducing the risk of an attacker discovering one random number and using it to circumvent the detection in another function. If an attacker does discover the value, the countermeasure will no longer be able to perform detection, but it will not be circumvented, because only the detection and not the security relies on it.

## 3.6 Related work[12]

Many countermeasures against code injection attacks exist. In this section, we briefly describe the different approaches that can be applied for protection against buffer overflows. The focus is on the countermeasures that are designed specifically to protect the stack from stack-smashing attacks.

### 3.6.1 Protection from attacks on stack-based vulnerabilities

Because the stack-based buffer overflow is a very widespread vulnerability, many countermeasures have been designed to protect against attacks on the stack. In this section we discuss the countermeasures that are most closely related to our countermeasure.

Two related countermeasures, StackGuard [49] and Propolice [62] were both discussed in Section 3.2. They rely on random values that must remain secret to provide protection.

[12]This section discusses work closely related to our countermeasure, a more extensive discussion on the topic of countermeasures for code injection attacks can be found in section 2.4

Stack Shield [172] is a countermeasure that attempts to protect against stack smashing attacks by copying the return address to another memory location, before entering the function call and restoring it just before returning from the function. This is an efficient countermeasure that protects the return address from attack, but still allows an attacker to use indirect pointer overwriting [34] to bypass the protection.

RAD [42] is similar to Stack Shield, except that it compares the return addresses stored at both locations and terminates the program if they are different. It solves some compatibility problems of Stack Shied and also better protects the area where the return addresses are copied to. However, it still only protect return addresses and, thus, could be bypassed using indirect pointer overwriting.

Xu et al. [191] suggest a similar approach to Stack Shield. Their countermeasure splits the stack into a control and a data stack. The control stack stores the return addresses while the data stack contains the rest of the data stored on the stack. Their implementation copies the return address to the control stack before entering the function call and copies it back from the control stack onto the data stack before returning from the function. The authors provide performance results for the SPEC CPU2000 benchmarks, the performance overheads associated with this approach range from 0.01% for 181.mcf to 23.77% for 255.vortex, which cane be prohibitively high for such a countermeasure.

Libverify [11] offers the same kind of protection as Stack Shield: upon entering a function it saves the return address on a return address stack (that it calls a canary stack) and when exiting from a function the saved return address is compared to the actual return address. The main difference with Stack Shield lies in the way that this check is added: Libverify does not require access to the source code of the application: the checks are added by dynamically linking the process with the library at run-time.

Libsafe [11] replaces the string manipulation functions that are prone to misuse with functions that prevent a buffer from being overflowed outside its stackframe. This is done by calculating the size of the input string and then making sure that the size of the source string is less than the upper bound of the destination string (the space from the variable's stack location to the saved frame pointer). If it is not smaller, the program is terminated. Once more, as is the case with several other countermeasures, this protection can be bypassed using indirect pointer overwriting.

Bhatkar et al. [20] incorporates a limited notion of our approach: it divides the stack into a safe stack (that only contains registers, return addresses and other values whose address is never taken) and a second stack for everything else.

### 3.6.2 Alternative approaches

Other approaches that protect against the more general problem of buffer overflows also protect against stack-based buffer overflows. In this section, we give a brief overview of the related work.

### Compiler-based countermeasures

Bounds checking [8, 89, 94, 110, 124, 144, 193] is the good solution for buffer overflows. However, performing bounds checking in C can have a severe impact on performance or may cause existing object code to become incompatible with bounds checked object code.

Protection of all pointers as provided by PointGuard [48] is an efficient implementation of a countermeasure that encrypts (using XOR) all pointers stored in memory with a randomly generated key and decrypts the pointer before loading it into a register. To protect the key, it is stored in a register upon generation and is never stored in memory. However, attackers could guess the decryption key if they are able to view several different encrypted pointers. Another attack described in [4] describes how an attacker could bypass PointGuard by partially overwriting a pointer. By only needing a partial overwrite, the randomness can be reduced, making a brute force attack feasible (if only one byte needs to be overwritten, the randomness is only 1 in 256,, instead of 1 in $2^{32}$ for four bytes).

### Operating system-based countermeasures

Non-executable memory [161, 170] tries to prevent code injection attacks by ensuring that the operating system does not allow execution of code that is not stored in the text segment of the program. This type of countermeasure can, however, be bypassed by a return-into-libc attack [188] where an attacker executes existing code (possibly with different parameters).

Randomized instruction sets [12, 93] also try to prevent an attacker from executing injected code by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. However, software-based implementations of this countermeasure incur large performance costs, while a hardware implementation is not immediately practical. Determined attackers may also be able to guess the encryption key and, as such, be able to inject code [163].

Address randomization [18, 170] is a technique that attempts to provide security by modifying the locations of objects in memory for different runs of a program. However, the randomization is limited in 32-bit systems (usually to 16 bits for the heap) and as a result may be inadequate for a determined attacker [151].

### Execution monitoring

In this section we describe two countermeasures that monitor the execution of a program and prevent control-flow transfers that could be unsafe.

Program shepherding [96] is a technique that monitors the execution of a program and disallows control-flow transfers[13] that are not considered safe. Program shepherding can be used for example to ensure that programs can only jump to entry points of functions or libraries, denying an attacker the possibility of bypassing checks that might be performed before a certain action is taken in a function. Program shepherding can also be used to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter which results in a performance impact that is significant for some programs, but acceptable for others.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does so by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. Performance overhead may be acceptable for some applications, but may be prohibitive for others.

## 3.7 Conclusion

In this document we described a countermeasure that protects against stack-based buffer overflows that has negligible performance overhead, while solving some of the shortcomings of existing efficient countermeasures. We assign all the different data types stored on the stack a high, medium or low ranking, both for the risk of it being an attack source and the value it has as a possible target. Using these rankings, we assign the data on the stack to different categories. Each of these categories is then mapped onto a separate stack. This effectively separates high-value targets from data that has a high risk of being used to launch an attack. A straight mapping of categories results in an implementation that has a very low performance overhead and offers better protection than existing countermeasures of comparable efficiency because it does not rely on random numbers. However, the memory usage in our implementation is higher than most other countermeasures, and we discuss possible ways to reduce it. One of the important advantages of our approach over existing approaches, is that it does not rely on the secrecy of canaries. Our countermeasure remains secure even if an attacker is able to read arbitrary memory locations because it is not based on random numbers.

[13]Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

# Chapter 4

# Improving memory management security for C and C++

*In this chapter, a paper is presented that improves the resilience of memory allocators against heap-based buffer overflow attacks. Often an attacker overwrites the management information that the memory manager stores with the dynamically allocated memory. By separating the management information from the regular data stored in this dynamic memory, it becomes harder for an attacker to perform a code injection attack when exploiting a heap-based buffer overflow. This countermeasure is very efficient and like the stack-based countermeasure, does not rely on secret random numbers to provide protection. As such, it is resilient against attackers who are able to read the address space of the program. This countermeasure was also a direct result of the more methodological approach discussed in [199], it is an application of the idea of separating control-flow data (data that is used to regulate the control flow of the program, like a return address) from regular data.*

*This paper is an extended version of work described in [200] that was presented in December 2006 at the Eighth International Conference on Information and Communication Security. It was written in collaboration with Wouter Joosen, Frank Piessens and Hans Vanden Eynden and was submitted to the Journal Of Computer Security for review in January 2007. A revised version was submitted in March 2008.*

## Abstract

Memory managers are an important part of any modern language: they are used to dynamically allocate memory for use in the program. Many managers exist and depending on the operating system and language. However, two major types of managers can be identified: manual memory allocators and garbage collectors. In the case of manual memory allocators, the programmer must manually release memory back to the system when it is no longer needed. Problems can occur when a programmer forgets to release it (memory leaks), releases it twice or keeps using freed memory. These problems are solved in garbage collectors. However, both manual memory allocators and garbage collectors store management information for the memory they manage. Often, this management information is stored where a buffer overflow could allow an attacker to overwrite this information, providing a reliable way to achieve code execution when exploiting these vulnerabilities. In this paper we describe several vulnerabilities for C and C++ and how these could be exploited by modifying the management information of a representative manual memory allocator and a garbage collector.

Afterwards, we present an approach that, when applied to memory managers, will protect against these attack vectors. We implemented our approach by modifying an existing widely used memory allocator. Benchmarks show that this implementation has a negligible, sometimes even beneficial, impact on performance.

## 4.1 Introduction

Security has become an important concern for all computer users. Worms and hackers are a part of every day Internet life. A particularly dangerous attack is the code injection attack, where attackers are able to insert code into the program's address space and can subsequently execute it. Programs written in C are particularly vulnerable to such attacks. Attackers can use a range of vulnerabilities to inject code. The most well known and most exploited is of course the standard buffer overflow: attackers write past the boundaries of a stack-based buffer and overwrite the return address of a function and point it to their injected code. When the function subsequently returns, the code injected by the attackers is executed [3].

These are not the only kind of code injection attacks though: a buffer overflow can also exist on the heap, allowing an attacker to overwrite heap-stored data. As pointers are not always available in normal heap-allocated memory, attackers often overwrite the management information that the memory manager relies upon to function correctly. A double free vulnerability, where a particular part of heap-allocated memory is deallocated twice could also be used by an attacker to inject code.

Many countermeasures have been devised that try to prevent code injection attacks [198]. However most have focused on preventing stack-based buffer overflows and only few have concentrated on protecting the heap or memory allocators from attack.

In this paper we evaluate a commonly used memory allocator and a garbage collector for C and C++ with respect to their resilience against code injection attacks and present a significant improvement for memory managers in order to increase robustness against code injection attacks. Our prototype implementation (which we call *dnmalloc*) comes at a very modest cost in both performance and memory usage overhead.

This paper is an extended version of work described in [200] which was presented in December 2006 at the Eighth International Conference on Information and Communication Security. The paper is structured as follows: Section 4.2 explains which vulnerabilities can exist for heap-allocated memory. Section 4.3 describes how both a popular memory allocator and a garbage collector can be exploited by an attacker using one of the vulnerabilities of Section 4.2 to perform code injection attacks. Section 4.4 describes our new more robust approach to handling the management information associated with chunks of memory. Section 4.5 contains the results of tests in which we compare our memory allocator to the original allocator in terms of performance overhead and memory usage. In Section 4.6 related work in improving security for memory allocators is discussed. Finally, Section 4.7 discusses possible future enhancements and presents our conclusion.

## 4.2 Heap-based vulnerabilities for code injection attacks[1]

There are a number of vulnerabilities that occur frequently and as such have become a favorite for attackers to use to perform code injection. We will examine how different memory allocators might be misused by using one of three common vulnerabilities: "heap-based buffer overflows", "off by one errors" and "dangling pointer references". In this section we will describe what these vulnerabilities are and how they could lead to a code injection attack.

### 4.2.1 Heap-based buffer overflow

Heap memory is dynamically allocated at run-time by the application. Buffer overflow, which are usually exploited on the stack, are also possible in this kind of memory. Exploitation of such heap-based buffer overflows usually relies on finding either function pointers or by performing an indirect pointer attack [34] on data pointers in this memory area. However, these pointers are not always present in the

[1]This section discusses the problem of heap-based buffer overflows, a more extensive discussion on the topic of vulnerabilities can be found in section 2.2

data stored by the program in this memory. As such, most attackers overwrite the memory management information that the memory allocator stores in or around memory chunks it manages. By modifying this information, attackers can perform an indirect pointer overwrite. This allows attackers to overwrite arbitrary memory locations, which could lead to a code injection attack [5, 195]. In the following sections we will describe how an attacker could use specific memory managers to perform this kind of attack.

### 4.2.2 Off by one errors

An off by one error is a special case of the buffer overflow. When an off by one occurs, the adjacent memory location is overwritten by exactly one byte. This often happens when a programmer loops through an array but typically ends at the array's size rather than stopping at the preceding element (because arrays start at 0). In some cases these errors can also be exploitable by an attacker [5, 195]. A more generally exploitable version of the off by one for memory allocators is an off by five, while these do not occur as often in the wild, they demonstrate that it is possible to cause a code injection attack when little memory is available. These errors are usually only exploitable on little endian machines because the least significant byte of an integer is stored before the most significant byte in memory.

### 4.2.3 Dangling pointer references

Dangling pointers are pointers to memory locations that are no longer allocated. In most cases dereferencing a dangling pointer will lead to a program crash. However in heap memory, it could also lead to a double free vulnerability, where a memory location is freed twice. Such a double free vulnerability could be misused by an attacker to modify the management information associated with a memory chunk and as a result could lead to a code injection attack [56]. This kind of vulnerability is not present in all memory managers, as some check whether a chunk is free or not before freeing it a second time. It may also be possible to write to memory which has already been reused, while the program think it is still writing to the original object. This can also lead to vulnerabilities. However, when such a vulnerability occurs, it is not always possible to reliably exploit these vulnerabilities as exploitation will most likely rely on the way the program uses the memory rather than by using the memory manager to the attacker's advantage.

In the following sections we will describe how a specific memory allocator could be exploited using dangling pointer references and more specifically, double free vulnerabilities. More information about these attacks can be found in [56, 195, 203].

## 4.3 Memory managers

In this section we will examine a representative memory allocator and a garbage collector for C and C++. We have chosen Doug Lea's memory allocator on which the Linux memory allocator is based, because this allocator is in wide use and illustrates typical vulnerabilities that are encountered in other memory allocators. A discussion of how other memory allocators can be exploited by attackers can be found in [203]. Boehm's garbage collector was chosen to determine whether a representative garbage collecting memory manager for C/C++ would be more resilient against attack.

We will describe how these memory managers work in normal circumstances and then will explain how a heap-vulnerability that can overwrite the management information of these memory managers could be used by an attacker to cause a code injection attack. We will use the same structure to describe both memory managers: first we describe how the manager works and afterwards we examine how an attacker could exploit it to perform code injection attacks (given one of the aforementioned vulnerabilities exists).

### 4.3.1 Doug Lea's memory allocator[2]

Doug Lea's memory allocator [107, 108] (commonly referred to as *dlmalloc*) was designed as a general-purpose memory allocator that can be used by any kind of program. *Dlmalloc* is used as the basis for *ptmalloc* [71], which is the allocator used in the GNU/Linux operating system. *Ptmalloc* mainly differs from dlmalloc in that it offers better support for multithreading, however this has no direct impact on the way an attacker can abuse the memory allocator's management information to perform code injection attacks. The description of *dlmalloc* in this section is based on version 2.7.2.

#### Description

The memory allocator divides the heap memory at its disposal into contiguous chunks[3], which vary in size as the various allocation routines (*malloc*, *free*, *realloc*, . . . ) are called. An invariant is that a free chunk never borders another free chunk when one of these routines has completed: if two free chunks had bordered, they would have been coalesced into one larger free chunk. These free chunks are kept in a doubly linked list, sorted by size. When the memory allocator at a later time requests a chunk of the same size as one of these free chunks, the first chunk of

[2]This section discusses Doug Lea's memory allocator and how heap-based vulnerabilities can be abused by attackers to perform code injection attacks when using this allocator, a more extensive discussion on the topic of vulnerabilities can be found in section 2.2

[3]A chunk is a block of memory that is allocated by the allocator, it can be larger than what a programmer requested because it usually reserves space for management information.

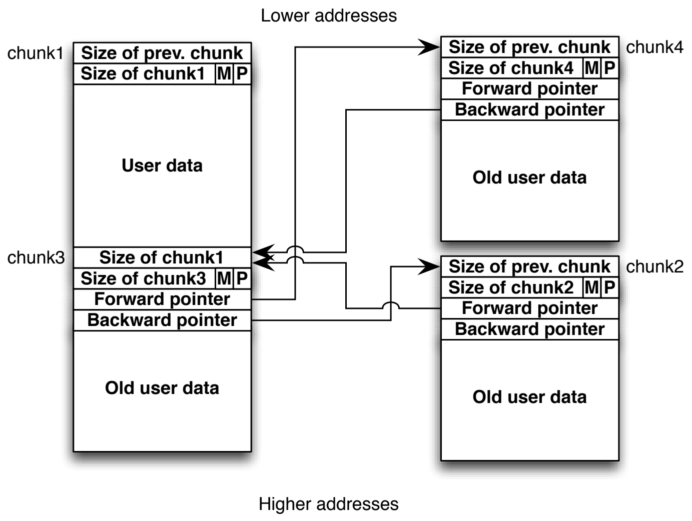


Figure 4.1: Heap containing used and free chunks

appropriate size will be removed from the list and made available for use in the program (i.e. it will turn into an allocated chunk).

**Chunk structure** Memory management information associated with a chunk is stored in-band. Figure 4.1 illustrates what a heap of used and unused chunks could look like. *Chunk1* is an allocated chunk containing information about the size of the chunk stored before it and its own size[4]. The rest of the chunk is available for the program to write data in. *Chunk3* is a free chunk that is allocated adjacent to *chunk1*. *Chunk2* and *chunk4* are free chunks located in arbitrary locations on the heap.

*Chunk3* is located in a doubly linked list together with *chunk2* and *chunk4*. *Chunk2* is the first chunk in the chain: its forward pointer points to *chunk3* and its backward pointer points to a previous chunk in the list. *Chunk3*'s forward pointer points to *chunk4* and its backward pointer points to *chunk2*. *Chunk4* is the last chunk in our example: its forward pointer points to a next chunk in the list and its backward pointer points to *chunk3*.

[4]The size of allocated chunks is always a multiple of eight, so the three least significant bits of the size field are used for management information: a bit to indicate if the previous chunk is in use (P) or not and one to indicate if the memory is mapped or not (M). The third bit is currently unused. The "previous chunk in use"-bit can be modified by an attacker to force coalescing of chunks. How this coalescing can be abused is explained later.

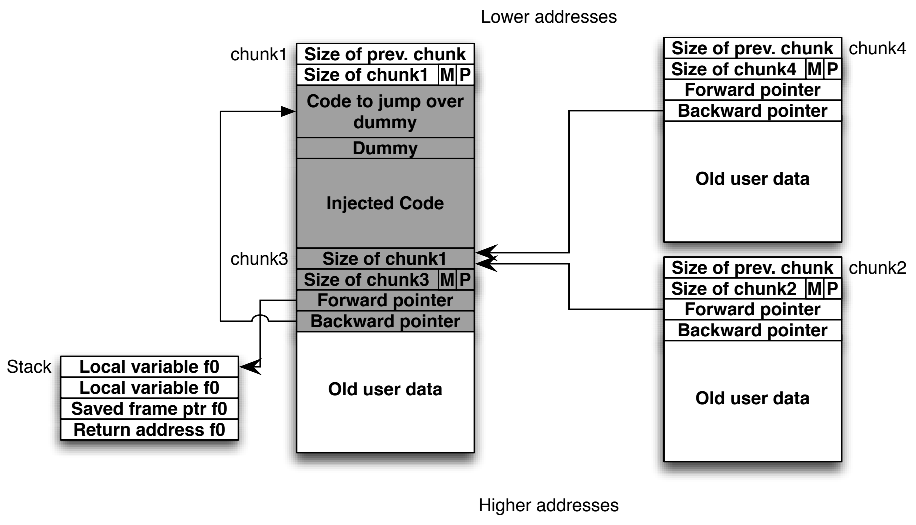


Figure 4.2: Heap-based buffer overflow in *dlmalloc*

## Exploitation

Dlmalloc is vulnerable to all three of the previously described vulnerabilities [5, 91, 162, 56]. Here we will describe how these vulnerabilities may lead to a code injection attack.

**Overwriting memory management information** Figure 4.2 shows what could happen if an array that is located in *chunk1* is overflowed: an attacker has overwritten the management information of *chunk3*. The size fields are left unchanged (although these could be modified if needed). The forward pointer has been changed to point to 12 bytes before the return address and the backward pointer has been changed to point to code that will jump over the next few bytes. When *chunk1* is subsequently freed, it will be coalesced together with *chunk3* into a larger chunk. As *chunk3* will no longer be a separate chunk after the coalescing it must first be removed from the list of free chunks.

The *unlink* macro takes care of this: internally a free chunk is represented by a struct containing the following unsigned long integer fields (in this order): *prev_size*, *size*, *forward* and *back*. A chunk is unlinked as follows:

Listing 4.1: Unlink macro

```
chunk2->forward->back = chunk2->back
chunk2->back->forward = chunk2->forward
```

Which is the same as (based on the struct used to represent malloc chunks):

Listing 4.2: Unlink macro expanded

```
*(chunk2->forward+12) = chunk2->back
*(chunk2->back+8) =  chunk2->forward
```

As a result, the value of the memory location that is twelve bytes after the location that *forward* points to will be overwritten with the value of *back*, and the value of the memory location eight bytes after the location that *back* points to will be overwritten with the value of *forward*. So in the example in Figure 4.2, the return address would be overwritten with a pointer to injected code. However, since the eight bytes after the memory that *back* points to will be overwritten with a pointer to *forward* (illustrated as dummy in Figure 4.2), the attacker needs to insert code to jump over the first twelve bytes into the first eight bytes of his injected code. Using this technique an attacker could overwrite arbitrary memory locations [5, 91, 162].

**Off by one error** An off by one error could also be exploited in the Doug Lea's memory allocator [5]. If the chunk is located immediately next to the next chunk (i.e., not padded to be a multiple of eight), then an off by one can be exploited: if the chunk is in use, the prev_size field of the next chunk will be used for data and by writing a single byte out of the bounds of the chunk, the least significant byte of the size field of the next chunk will be overwritten. As the least significant byte contains the prev_inuse bit, the attacker can make the allocator think the chunk is free and will coalesce it when the second chunk is freed. Figure 4.3 depicts the exploit: the attacker creates a fake chunk in the *chunk1* and sets the prev_size field accordingly and overwrites the least significant byte of *chunk2*'s size field to mark the current chunk as free. The same technique using the forward and backward pointers (in the fake chunk) that was used in Section 4.3.1 can now be used to overwrite arbitrary memory locations.

**Double free** Dlmalloc can be used for a code injection attack if a double free exists in the program [56]. Figure 4.4 illustrates what happens when a double free occurs. The full lines in this figure are an example of what the list of free chunks of memory might look like when using this allocator.

*Chunk1* is larger than *chunk2* and *chunk3* (which are both the same size), meaning that *chunk2* is the first chunk in the list of free chunks of equal size. When a new chunk of the same size as *chunk2* is freed, it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of *chunk1* and the forward pointer of *chunk2*.

When a chunk is freed twice it overwrites the forward and backward pointers and could allow an attacker to overwrite arbitrary memory locations at some later

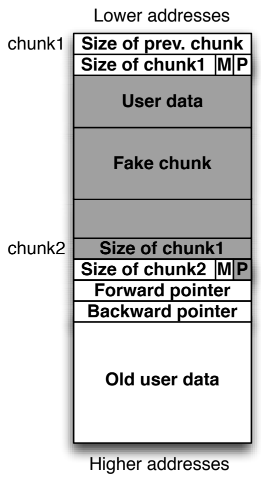


Figure 4.3: Off by one error in *dlmalloc*

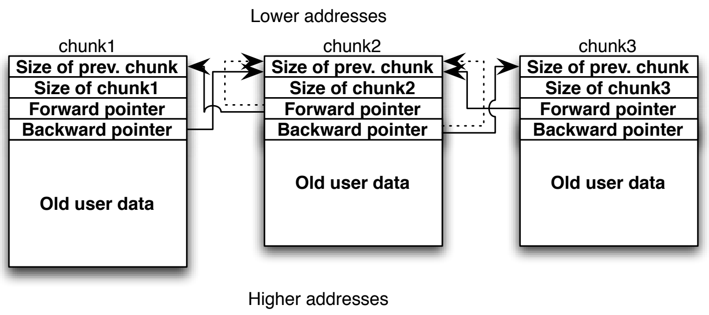


Figure 4.4: List of free chunks in dlmalloc: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred.

point in the program. As mentioned in the previous section: if a new chunk of the same size as *chunk2* is freed it will be placed before *chunk2* in the list. The following pseudo code demonstrates this (modified from the original version found in dlmalloc):

Listing 4.3: Adding a chunk to the list of free chunks

```
tmpback = front_of_list_of_same_size_chunks
tmpforward = tmpback->forward
new_chunk->back = tmpback
new_chunk->forward = tmpforward
tmpforward->back = tmpback->forward = new_chunk
```

The backward pointer of *new_chunk* is set to point to *chunk2*, the forward pointer of this backward pointer (i.e., *chunk2—>forward = chunk1*) will be set as the forward pointer for *new_chunk*. The backward pointer of the forward pointer (i.e., *chunk1—>back*) will be set to *new_chunk* and the forward pointer of the backward pointer (*chunk2—>forward*) will be set to *new_chunk*.

If chunk2 would be freed twice in succession, the following would happen (substitutions made on the code listed above):

Listing 4.4: Freeing chunk2 twice

```
back = chunk2
forward = chunk2->forward
chunk2->back = chunk2
chunk2->forward = chunk2->forward
chunk2->forward->back = chunk2->forward = chunk2
```

The forward and backward pointers of *chunk2* both point to itself. The dotted lines in Figure 4.4 illustrate what the list of free chunks looks like after a second free of *chunk2*.

Listing 4.5: Unlinking chunk2

```
chunk2->forward->back = chunk2->back
chunk2->back->forward = chunk2->forward
```

But since both *chunk2—>forward* and *chunk2—>back* point to *chunk2*, it will again point to itself and will not really be unlinked. However the allocator assumes it has and the program is now free to use the user data part (everything below 'size of chunk' in Figure 4.4) of the chunk for its own use.

Attackers can now use the same technique that we previously discussed to exploit the heap-based overflow (see Figure 4.2): they set the forward pointer to point 12 bytes before the return address and change the value of the backward pointer to point to code that will jump over the bytes that will be overwritten. When the program tries to allocate a chunk of the same size again, it will again try to unlink *chunk2*, which will overwrite the return address with the value of *chunk2's* backward pointer.

### 4.3.2 Boehm garbage collector

The Boehm garbage collector [29, 28, 27] is a conservative garbage collector [5] for C and C++ that can be used instead of *malloc* or *new*. Programmers can request memory without having to explicitly free it when they no longer need it. The garbage collector will automatically release memory to the system when it is no longer needed. If the programmer does not interfere with memory that is managed by the garbage collector (explicit deallocation is still possible), dangling pointer references are made impossible.

#### Description

Memory is allocated by the programmer by a call to *GC_malloc* with a request for a number of bytes to allocate. The programmer can also explicitly free memory using *GC_free* or can resize the chunk by using *GC_realloc*. These two calls could however lead to dangling pointer references.

**Memory structure** The collector makes a difference between large and small chunks. Large chunks are larger than half of the value of HBLKSIZE[6]. These large chunks are rounded up to the next multiple of HBLKSIZE and allocated. When a small chunk is requested and none are free, the allocator will request HBLKSIZE memory from the system and divide it in small chunks of the requested size.

There is no special structure for an allocated chunk, it only contains data. A free chunk contains a pointer at the beginning of the chunk that points to the next free chunk to form a linked list of free chunks of a particular size.

**Collection modes** The garbage collector has two modes: incremental and non-incremental modes. In incremental mode, the heap will be increased in size whenever insufficient space is available to fulfill an allocation request. Garbage collection only starts when a certain threshold of heap size is reached. In non-incremental mode whenever a memory allocation would fail without resizing the heap the

[5]A conservative collector assumes that each memory location is a pointer to another object if it contains a value that goes into an allocated chunk of memory. This can result in false negatives where some memory is incorrectly identified as still being allocated.

[6]HBLKSIZE is equal to page size on IA32.

garbage collector decides (based on a threshold value) whether or not to start collecting.

**Collection** Collection is done using a mark and sweep algorithm. This algorithm works in three steps. First all objects are marked as being unreachable (i.e., candidates to be freed). The allocator then starts at the roots (registers, stack, static data) and iterates over every pointer that is reachable starting from one of these objects. When an object is reachable it is marked accordingly. Afterwards the removal phase starts: large unreachable chunks are placed in a linked list and large adjacent chunks are coalesced. Pages containing small chunks are also examined: if all of the chunks on the page are unreachable, the entire page is placed in the list of large chunks. If it is not free, the small chunks are placed in a linked list of small chunks of the same size.

### Exploitation

Although the garbage collector removes vulnerabilities like dangling pointer references, it is still vulnerable to buffer overflows. It is also vulnerable to a double free vulnerability if the programmer explicitly frees memory.

**Overwriting memory management information** During the removal phase, objects are placed in a linked list of free chunks of the same size that is stored at the start of the chunk. If attackers can write out of the boundaries of a chunk, they can overwrite the pointer to the next chunk in the linked list and make it refer to the target memory location. When the allocator tries to reallocate a chunk of the same size it will return the memory location as a chunk and as a result will allow the attacker to overwrite the target memory location.

**Off by five** The garbage collector will automatically add padding to an object to ensure that the property of C/C++ which allows a pointer to point to one element past an array is recognized as pointing to the object rather than the next. This padding forces an attacker to overwrite the padding (4 bytes on IA32). He can then overwrite the first four bytes of the next chunk with an off by eight attack. If the target memory location is located close to a chunk and only the least significant byte of the pointer needs to be modified then an off by five might suffice.

**Double free** Dangling pointer references cannot exist if the programmer does not interfere with the garbage collector. However if the programmer explicitly frees memory, a double free can occur and could be exploitable.

Figures 4.5 and 4.6 illustrate how this vulnerability can be exploited: *chunk1* was the last chunk freed and was added to the start of the linked list and points to *chunk2*. If *chunk2* is freed a second time it will be placed at the beginning of the list, but *chunk1* will still point to it. When *chunk2* is subsequently reallocated, it will be writable and still be located in the list of free chunks. The location where the pointer resides is now writable by the program. If attackers control what is written to the chunk, they can modify the pointer and if more chunks of the same size are allocated eventually the chunk to which *chunk2* points will be returned as a valid chunk, allowing the attackers to overwrite arbitrary memory locations.

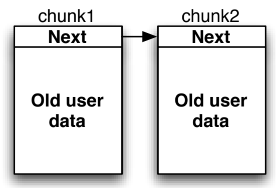


Figure 4.5: Linked list of free chunks in Boehm's garbage collector

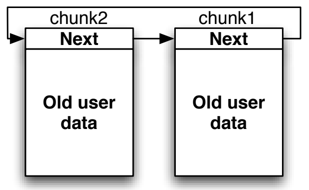


Figure 4.6: Double free of *chunk2* in Boehm's garbage collector

### 4.3.3 Summary

The memory allocator we presented in this section is representative for the many memory allocators that are in common use today. There are many others like the memory allocator used by Windows, the allocator used in the Solaris and IRIX operating systems or the allocator used in FreeBSD that are also vulnerable to similar attacks [99, 5, 13].

In the previous section we also discussed how a garbage collector can be vulnerable to the same attacks that are often performed on memory allocators.

## 4.4 A more secure memory allocator

As can be noted from the previous sections many memory managers are vulnerable to code injection attacks if an attacker can modify its management information. In this section we describe a new approach to handling the management information that is more robust against these kind of attacks. This new approach could be applied to the managers discussed above and we also describe a prototype implementation (called *dnmalloc*) where we modified *dlmalloc* to incorporate the changes we described.

### 4.4.1 Countermeasure Design

The main principle used to design this countermeasure is to separate management information (*chunkinfo*) from the data stored by the user (*chunkdata*). This management information is then stored in separate contiguous memory regions that only contain other management information. To protect these regions from being overwritten by overflows in other memory mapped areas, they are protected by guard pages. This simple design essentially makes overwriting the *chunkinfo* by using a heap-based buffer overflow impossible. Figure 4.7 depicts the typical memory layout of a program that uses a general memory allocator (on the left) and one that uses our modified design (on the right)

Most memory allocators allocate memory in the datasegment that can be increased (or decreased) as necessary using the *brk* systemcall [166]. However, when larger chunks are requested, it can also allocate memory in the shared memory area [7] using the *mmap*[8] systemcall to allocate memory for the chunk. In Figure 4.7, we have depicted this behavior: there are chunks allocated in both the heap and in the shared memory area. Note that a program can also map files and devices into this region itself, we have depicted this in Figure 4.7 in the boxes labeled *'Program mapped memory'*.

In this section we describe the structures needed to perform this separation in a memory allocator efficiently. In the next paragraph we describe the structures that are used to retrieve the *chunkinfo* when presented with a pointer to *chunkdata*. In the paragraph that follows the next, we discuss the management of the region where these *chunkinfos* are stored.

#### Lookup table and lookup function

To perform the separation of the management information from the actual *chunkdata*, we use a *lookup table*. The entries in the *lookup table* contain pointers to the *chunkinfo* for a particular *chunkdata*. When given such a *chunkdata* address, a lookup function is used to find the correct entry in the *lookup table*.

The table is stored in a map of contiguous memory that is big enough to hold the maximum size of the *lookup table*. This map can be large on 32-bit systems, however it will only use virtual address space rather than physical memory[9]. Physical memory will only be allocated by the operating system when the specific page is written to. To protect this memory from buffer overflows in other memory in the shared memory region, a guard page is placed before it. At the right hand

[7]Note that memory in this area is not necessarily shared among applications, it has been allocated by using *mmap*

[8]mmap is used to map files or devices into memory. However, when passing it the *MAP_ANON* flag or mapping the */dev/zero* file, it can be used to allocate a specific region of contiguous memory for use by the application (however, the granularity is restricted to page size) [166].

[9]While this will not cause physical memory overhead, the use of a large amount of virtual address space will use up more page table entries, resulting in performance overhead.

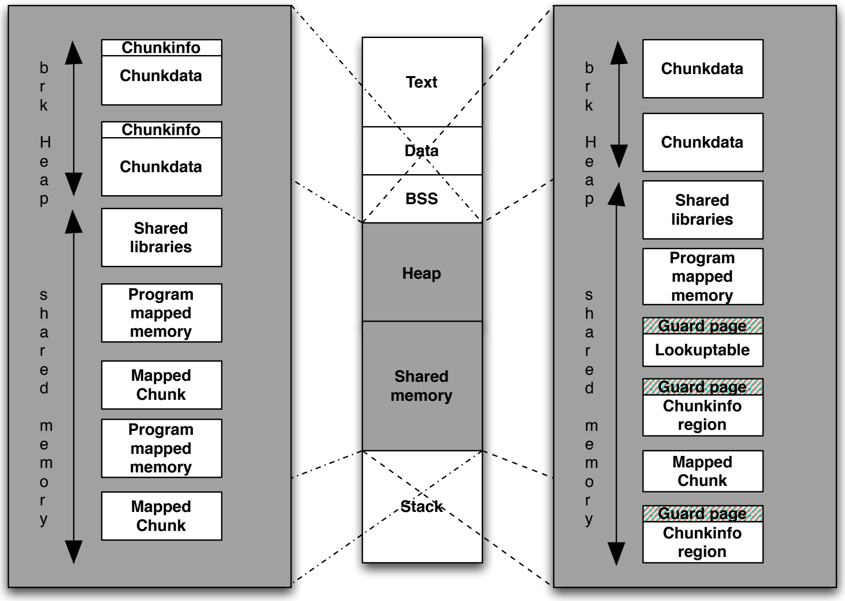


Figure 4.7: Original (left) and modified (right) process memory layout

side of Figure 4.7 we illustrate what the layout looks like in a typical program that uses this design.

### Chunkinfo regions

*Chunkinfos* are also stored in a particular contiguous region of memory (called a *chunkinfo region*), which is protected from other memory by a guard page. This region also needs to be managed, several options are available for doing this. We will discuss the advantages and disadvantages of each.

Our preferred design, which is also the one used in our implementation and the one depicted in Figure 4.7, is to map a region of memory large enough to hold a predetermined amount of *chunkinfos*. To protect its contents, we place a guard page at the top of the region. When the region is full, a new region, with its own guard page, is mapped and added to a linked list of *chunkinfo regions*. This region then becomes the active region, meaning that all requests for new *chunkinfos* that cannot be satisfied by existing *chunkinfos*, will be allocated in this region. The disadvantage of this technique is that a separate guard page is needed for every *chunkinfo region*, because the allocator or program may have stored data in the same region (as depicted in Figure 4.7). Although such a guard page does not need actual memory (it will only use virtual address space), setting the correct permissions for it is an expensive system call (requiring the system to perform several time-consuming actions to execute).

When a *chunkdata* disappears, either because the associated memory is released back to the system or because two *chunkdatas* are coalesced into one, the *chunkinfo* is stored in a linked list of free *chunkinfos*. In this design, we have a separate list of free *chunkinfos* for every region. This list is contained in one of the fields of the *chunkinfo* that is unused because it is no longer associated with a *chunkdata*. When a new *chunkinfo* is needed, the allocator returns one of these free *chunkinfos*: it goes over the lists of free *chunkinfos* of all existing *chunkinfo regions* (starting at the currently active region) to attempt to find one. If none can be found, it allocates a new *chunkinfo* from the active region. If all *chunkinfos* for a region have been added to its list of free *chunkinfos*, the entire region is released back to the system.

An alternative design is to map a single *chunkinfo region* into memory large enough to hold a specific amount of *chunkinfos*. When the map is full, it can be extended as needed. The advantage is that there is one large region, and as such, not much management is required on the region, except growing and shrinking it as needed. This also means that we only need a single guard page at the top of the region to protect the entire region. However, a major disadvantage of this technique is that, if the virtual address space behind the region is not free, extension means moving it somewhere else in the address space. While the move operation is not expensive because of the paging system used in modern operating systems, it invalidates the pointers in the *lookup table*. Going over the entire *lookup* *table* and modifying the pointers is prohibitively expensive. A possible solution to this is to store offsets in the *lookup table* and to calculate the actual address of the *chunkinfo* based on the base address of the *chunkinfo region*.

A third design is to store the *chunkinfo region* directly below the maximum size the stack can grow to (if the stack has such a fixed maximum size), and make the *chunkinfo region* grow down toward the heap. This eliminates the problem of invalidation as well, and does not require extra calculations to find a *chunkinfo*, given an entry in the *lookup table*. To protect this region from being overwritten by data stored on the heap, a guard page has to be placed at the top of the region, and has to be moved every time the region is extended. A major disadvantage of this technique is that it can be hard to determine the start of the stack region on systems that use address space layout randomization [170]. It is also incompatible with programs that do not have a fixed maximum stack size.

These last two designs only need a single, but sorted, list of free *chunkinfos*. When a new *chunkinfo* is needed, it can return, respectively, the lowest or highest address from this list. When the free list reaches a predetermined size, the region can be shrunk and the active *chunkinfos* in the shrunk area are copied to free space in the remaining *chunkinfo region*.

### 4.4.2 Prototype Implementation

*Dnmalloc* was implemented by modifying *dlmalloc 2.7.2* to incorporate the changes described in Section 4.4.1. The ideas used to build this implementation, however, could also be applied to other memory allocators. *Dlmalloc* was chosen because it is very widely used (in its *ptmalloc* incarnation) and is representative for this type of memory allocators. *Dlmalloc* was chosen over *ptmalloc* because it is less complex to modify and because the modifications done to *dlmalloc* to achieve *ptmalloc* do not have a direct impact on the way the memory allocator can be abused by an attacker.

#### Lookup table and lookup function

The *lookup table* is in fact a lightweight hashtable: to implement it, we divide every page in 256 possible chunks of 16 bytes (the minimum chunk size), which is the maximum amount of chunks that can be stored on a single page in the heap. These 256 possible chunks are then further divided into 32 groups of 8 elements. For every such group we have 1 entry in the *lookup table* that contains a pointer to a linked list of these elements (which has a maximum size of 8 elements). As a result we have a maximum of 32 entries for every page. The *lookup table* is allocated using the memory mapping function, mmap. This allows us to reserve virtual address space for the maximum size that the *lookup table* can become without using physical memory. Whenever a new page in the *lookup table* is accessed, the operating system will allocate physical memory for it.

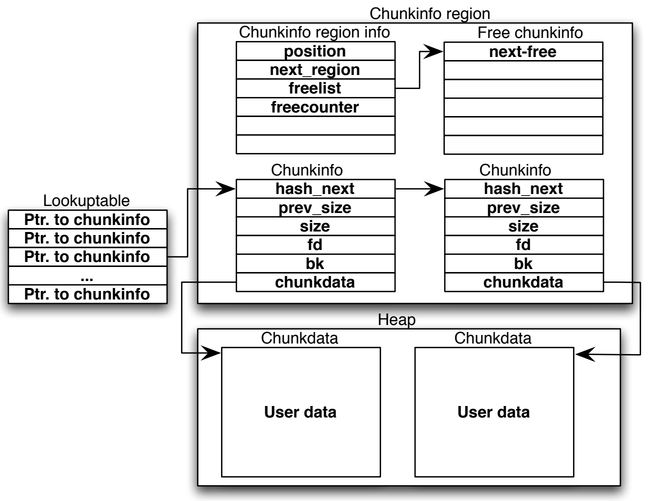


Figure 4.8: *Lookup table* and chunkinfo layout

We find an entry in the table for a particular group from a *chunkdata*'s address in two steps:

1. We subtract the address of the start of the heap from the *chunkdata*'s address.

2. Then we shift the resulting value 7 bits to the right. This will give us the entry of the chunk's group in the *lookup table*.

To find the *chunkinfo* associated with a chunk we now have to go over a linked list that contains a maximum of 8 entries and compare the *chunkdata*'s address with the pointer to the *chunkdata* that is stored in the *chunkinfo*. This linked list is stored in the hashnext field of the *chunkinfo* (illustrated in Figure 4.8).

### Chunkinfo

A *chunkinfo* contains all the information that is available in *dlmalloc*, and adds several extra fields to correctly maintain the state. The layout of a *chunkinfo* is illustrated in Figure 4.8: the *prev_size*, *size*, *forward* and *backward* pointers serve the same purpose as they do in *dlmalloc*, the *hashnext* field contains the linked list that we mentioned in the previous section and the *chunkdata* field contains a pointer to the actual allocated memory.

### 4.4.3 Managing chunk information

The chunk information itself is stored in a fixed map that is big enough to hold a predetermined amount of *chunkinfos*. Before this area a guard page is mapped, to prevent the heap from overflowing into this memory region. Whenever a new *chunkinfo* is needed, we simply allocate the next 24 bytes in the map for the *chunkinfo*. When we run out of space, a new region is mapped together with a guard page.

One *chunkinfo* in the region is used to store the meta-data associated with a region. This metadata (illustrated in Figure 4.8, by the *chunkinfo region info* structure) contains a pointer to the start of the list of free chunks in the freelist field. It also holds a counter to determine the current amount of free *chunkinfos* in the region. When this number reaches the maximum amount of chunks that can be allocated in the region, it will be deallocated. The *chunkinfo region info* structure also contains a position field that determines where in the region to allocate the next *chunkinfo*. Finally, the next_region field contains a pointer to the next *chunkinfo* region.

## 4.5 Evaluation

The realization of these extra modifications comes at a cost: both in terms of performance and in terms of memory overhead. To evaluate how high the performance overhead of *dnmalloc* is compared to the original *dlmalloc*, we ran the full SPEC® CPU2000 Integer reportable benchmark [79], which gives us an idea of the overhead associated with general-purpose single-threaded programs. We also evaluated the implementation using a suite of allocator-intensive benchmarks, which have been widely used to evaluate the performance of memory managers [75, 88, 16, 17]. While these two suites of benchmarks make up the macrobenchmarks of this section, we also performed microbenchmarks to get a better understanding of which allocator functions are faster or slower when using *dnmalloc*.

Table 4.1 holds a description of the programs that were used in both the macro- and the microbenchmarks. For all the benchmarked applications we have also included the number of times they call the most important memory allocation functions: *malloc*, *realloc*, *calloc*[10] and free (the SPEC® benchmark calls programs multiple times with different inputs for a single run; for these we have taken the average number of calls).

The results of the performance evaluation can be found in Section 4.5.1. Both macrobenchmarks and the microbenchmarks were also used to measure the memory overhead of our prototype implementation compared to *dlmalloc*. In Section 4.5.2 we discuss these results. Finally, we also performed an evaluation of the se-

[10]This memory allocator call will allocate memory and will then clear it by ensuring that all memory is set to 0

| **SPEC CPU2000 Integer benchmark programs** | | | | | |
|---|---|---|---|---|---|
| Program | Description | malloc | realloc | calloc | free |
| 164.gzip | Compression utility | 87,241 | 0 | 0 | 87,237 |
| 175.vpr | FPGA placement routing | 53,774 | 9 | 48 | 51,711 |
| 176.gcc | C compiler | 22,056 | 2 | 0 | 18,799 |
| 181.mcf | Network flow solver | 2 | 0 | 3 | 5 |
| 186.crafty | Chess program | 39 | 0 | 0 | 2 |
| 197.parser | Natural language processing | 147 | 0 | 0 | 145 |
| 252.eon | Ray tracing | 1,753 | 0 | 0 | 1,373 |
| 253.perlbmk | Perl | 4,412,493 | 195,074 | 0 | 4,317,092 |
| 254.gap | Computational group theory | 66 | 0 | 1 | 66 |
| 255.vortex | Object Oriented Database | 6 | 0 | 1,540,780 | 1,467,029 |
| 256.bzip2 | Compression utility | 12 | 0 | 0 | 2 |
| 300.twolf | Place/route simulator | 561,505 | 4 | 13,062 | 492,727 |
| **Allocator-intensive benchmarks** | | | | | |
| Program | Description | malloc | realloc | calloc | free |
| boxed-sim | Balls-in-box simulator | 3,328,299 | 63 | 0 | 3,312,113 |
| cfrac | Factors numbers | 581,336,282 | 0 | 0 | 581,336,281 |
| espresso | Optimizer for PLAs | 5,084,290 | 59,238 | 0 | 5,084,225 |
| lindsay | Hypercube simulator | 19,257,147 | 0 | 0 | 19,257,147 |

Table 4.1: Programs used in the evaluations of *dnmalloc*

| **SPEC CPU2000 Integer benchmark programs** | | | |
|---|---|---|---|
| Program | Dlmalloc r/t (s) | Dnmalloc r/t (s) | R/t overhead |
| 164.gzip | 253 ± 0 | 253 ± 0 | 0% |
| 175.vpr | 361 ± 0.15 | 361.2 ± 0.14 | 0.05% |
| 176.gcc | 153.9 ± 0.05 | 154.1 ± 0.04 | 0.13% |
| 181.mcf | 287.3 ± 0.07 | 290.1 ± 0.07 | 1% |
| 186.crafty | 253 ± 0 | 252.9 ± 0.03 | -0.06% |
| 197.parser | 347 ± 0.01 | 347 ± 0.01 | 0% |
| 252.eon | 770.3 ± 0.17 | 782.6 ± 0.1 | 1.6% |
| 253.perlbmk | 243.2 ± 0.04 | 255 ± 0.01 | 4.86% |
| 254.gap | 184.1 ± 0.03 | 184 ± 0 | -0.04% |
| 255.vortex | 250.2 ± 0.04 | 223.6 ± 0.05 | -10.61% |
| 256.bzip2 | 361.7 ± 0.05 | 363 ± 0.01 | 0.35% |
| 300.twolf | 522.9 ± 0.44 | 511.9 ± 0.55 | -2.11% |
| **Allocator-intensive benchmarks** | | | |
| Program | Dlmalloc r/t (s) | Dnmalloc r/t (s) | R/t overhead |
| boxed-sim | 230.6 ± 0.08 | 232.2 ± 0.12 | 0.73% |
| cfrac | 552.9 ± 0.05 | 587.9 ± 0.01 | 6.34% |
| espresso | 60 ± 0.02 | 60.3 ± 0.01 | 0.52% |
| lindsay | 239.1 ± 0.02 | 242.3 ± 0.02 | 1.33% |

Table 4.2: Average macrobenchmark runtime results for *dlmalloc* and *dnmalloc*

curity of *dnmalloc* in Section 4.5.3 by running a set of exploits against real world programs using both *dlmalloc* and *dnmalloc*.

*Dnmalloc* and all files needed to reproduce these benchmarks are available publicly [196].

### 4.5.1 Performance

This section evaluates our countermeasure in terms of performance overhead. All benchmarks were run on 10 identical machines (Pentium 4 2.80 Ghz, 512MB RAM, no hyperthreading, Redhat 6.2, kernel 2.6.8.1).

#### Macrobenchmarks

To perform these benchmarks, the SPEC® benchmark was run 10 times on these PCs for a total of 100 runs for each allocator. The allocator-intensive benchmarks were run 50 times on the 10 PCs for a total of 500 runs for each allocator.

Table 4.2 contains the average runtime (denoted as r/t in the table), including standard error, of the programs in seconds. The results show that the runtime

| **Microbenchmarks** | | | |
|---|---|---|---|
| Program | Dlmalloc r/t (s) | Dnmalloc r/t (s) | R/t Overh. |
| no memset: malloc | 0.28721 ± 0.00108 | 0.06488 ± 0.00007 | -77.41% |
| no memset: realloc | 1.99831 ± 0.00055 | 1.4608 ± 0.00135 | -26.9% |
| no memset: free | 0.06737 ± 0.00001 | 0.03691 ± 0.00001 | -45.21% |
| no memset: calloc | 0.32744 ± 0.00096 | 0.2142 ± 0.00009 | -34.58% |
| with memset: malloc | 0.32283 ± 0.00085 | 0.39401 ± 0.00112 | 22.05% |
| with memset: realloc | 2.11842 ± 0.00076 | 1.26672 ± 0.00105 | -40.2% |
| with memset: free | 0.06754 ± 0.00001 | 0.03719 ± 0.00005 | -44.94% |
| with memset: calloc | 0.36083 ± 0.00111 | 0.1999 ± 0.00004 | -44.6% |

Table 4.3: Average microbenchmark runtime results for *dlmalloc* and *dnmalloc*

overhead of our allocator are mostly negligible both for general programs as for allocator-intensive programs. However, for *perlbmk* and *cfrac* the performance overhead is slightly higher: 4% and 6%. These show that even for such programs the overhead for the added security is extremely low. In some cases (*vortex* and *twolf*) the allocator even improves performance. This is mainly because of improved locality of management information in our approach: in general all the management information for several chunks will be on the same page, which results in more cache hits [75]. When running the same tests on the same system with L1 and L2 cache[11] disabled, the performance benefit for *vortex* went down from 10% to 4.5%.

### Microbenchmarks

Listing 4.6: Example microbenchmark (no memset)

```
gettimeofday(&tv1, 0);

while (i<LOOPS) {
 randm = random() % 4081;
 memarray[i] = malloc(basesize + randm);
 i++;
}

gettimeofday(&tv2, 0);
```

[11]These are caches that are faster than the actual memory in a computer and are used to reduce the cost of accessing general memory [171].

```
d = (double)((tv2.tv_sec - tv1.tv_sec) * 1000000
+ (tv2.tv_usec - tv1.tv_usec)) / 1000000;
```

We have included two microbenchmarks. In the first microbenchmark, the time that the program takes to perform 100,000 *mallocs* of random[12] chunk sizes ranging between 16 and 4096 bytes was measured (see Listing 4.6 for a code snippet). Afterwards the time was measured for the same program to *realloc* these chunks to different random size (also ranging between 16 and 4096 bytes). We then measured how long it took the program to *free* those chunks and finally to *calloc* 100,000 new chunks of random sizes. The second benchmark does essentially the same but also performs a *memset*[13] on the memory it allocates (using *malloc*, *realloc* and *calloc*). The microbenchmarks were each run 100 times on a single PC (the same configuration as was used for the macrobenchmarks) for each allocator.

The average of the results (in seconds) of these benchmarks, including the standard error, for *dlmalloc* and *dnmalloc* can be found in Table 4.3. Although it may seem from the results of the *loop* program that the *malloc* call has an enormous speed benefit when using *dnmalloc*, this is mainly because our implementation does not access the memory it requests from the system. This means that on systems that use optimistic memory allocation (which is the default behavior on Linux) our allocator will only use memory when the program accesses it.

To measure the actual overhead of our allocator when the memory is accessed by the application, we also performed the same benchmark in the program *loop2*, but in this case always set all bytes in the acquired memory to a specific value. Again there are some caveats in the measured result: while it may seem that our *calloc* function is much faster, in fact it has the same overhead as the *malloc* function followed by a call to *memset* (because *calloc* will call *malloc* and then set all bytes in the memory to 0). However, the place where it is called in the program is of importance here: it was called after a significant amount of chunks were freed and as a result this call will reuse existing free chunks. Calling *malloc* in this case would have produced similar results.

The main conclusion we can draw from these microbenchmarks is that the performance of our implementation is very close to that of *dlmalloc*: it is faster for some operations, but slower for others.

### 4.5.2 Memory overhead

Our implementation also has an overhead when it comes to memory usage: the original allocator has an overhead of approximately 8 bytes per chunk. Our implementation has an overhead of approximately 24 bytes to store the chunk information and for every 8 chunks, a *lookup table* entry will be used (4 bytes).

[12]A fixed seed was set so two runs of the program return the same results

[13]This call will fill a particular range in memory with a particular byte.

Depending on whether the chunks that the program uses are large or small, our overhead could be low or high. To test the memory overhead on real world programs, we measured the memory overhead for the benchmarks we used to test performance, the results (in megabytes) can be found in Table 4.4. They contain the complete overhead of all extra memory the countermeasure uses compared to *dlmalloc*.

| **SPEC CPU2000 Integer benchmark programs** | | | |
|---|---|---|---|
| Program | dlmalloc mem. use (MB) | our mem. use (MB) | Overhead |
| 164.gzip | 180.37 | 180.37 | 0% |
| 175.vpr | 20.07 | 20.82 | 3.7% |
| 176.gcc | 81.02 | 81.14 | 0.16% |
| 181.mcf | 94.92 | 94.92 | 0% |
| 186.crafty | 0.84 | 0.84 | 0.12% |
| 197.parser | 30.08 | 30.08 | 0% |
| 252.eon | 0.33 | 0.34 | 4.23% |
| 253.perlbmk | 53.80 | 63.37 | 17.8% |
| 254.gap | 192.07 | 192.07 | 0% |
| 255.vortex | 60.17 | 63.65 | 5.78% |
| 256.bzip2 | 184.92 | 184.92 | 0% |
| 300.twolf | 3.22 | 5.96 | 84.93% |
| **Allocator-intensive benchmarks** | | | |
| Program | dlmalloc mem. use (MB) | our mem. use (MB) | Overhead |
| boxed-sim | 0.78 | 1.16 | 49.31% |
| cfrac | 2.14 | 3.41 | 59.13% |
| espresso | 5.11 | 5.88 | 15.1% |
| lindsay | 1.52 | 1.57 | 2.86% |
| **Microbenchmarks** | | | |
| loop/loop2 | 213.72 | 217.06 | 1.56% |

Table 4.4: Average memory usage for *dlmalloc* and *dnmalloc*

In general, the relative memory overhead of our countermeasure is fairly low (generally below 20%), but in some cases the relative overhead can be very high, this is the case for *twolf*, *boxed-sim* and *cfrac*. These applications use many very small chunks, so while the relative overhead may seem high, if we examine the absolute overhead it is fairly low (ranging from 120 KB to 2.8 MB). Applications that use larger chunks have a much smaller relative memory overhead.

| Exploit for | Dlmalloc | Dnmalloc |
|---|---|---|
| Wu-ftpd 2.6.1 [204] | Shell | Continues |
| Sudo 1.6.1 [92] | Shell | Crash |
| Sample heap-based buffer overflow | Shell | Continues |
| Sample double free | Shell | Continues |

Table 4.5: Results of exploits against vulnerable programs protected with *dnmalloc*

### 4.5.3 Security evaluation

In this section we present experimental results when using our memory allocator to protect applications with known vulnerabilities against existing exploits.

Table 4.5 contains the results of running several exploits against known vulnerabilities when these programs were compiled using *dlmalloc* and *dnmalloc* respectively. When running the exploits against *dlmalloc*, we were able to execute a code injection attack in all cases. However, when attempting to exploit *dnmalloc*, the overflow would write into adjacent chunks, but would not overwrite the management information, as a result, the programs kept running.

These kinds of security evaluations can only prove that a particular attack works, but it cannot disprove that no variation of this attack exists that does work. Because of the fragility of exploits, a simple modification in which an extra field is added to the memory management information for the program would cause many exploits to fail. While this is useful against automated attacks, it does not provide any real protection from a determined attacker. Testing exploits against a security solution can only be used to prove that it can be bypassed. As such, we provide these evaluations to demonstrate how our countermeasure performs when confronted with a real world attack, but we do not make any claims as to how accurately they evaluate the security benefit of *dnmalloc*.

However, the design in itself of the allocator gives strong security guarantees against buffer overflows, since none of the memory management information is stored with user data. We contend that it is impossible to overwrite it using a heap-based buffer overflow. If such an overflow occurs, an attacker will start at a chunk and will be able to overwrite any data that is behind it. Since such an buffer overflow is contiguous, the attacker will not be able to overwrite the management information. If an attacker is able to write until the management information, it will be protected by the guard page. An attacker could use a pointer stored in heap memory to overwrite the management information, but this would be a fairly useless operation: the management information is only used to be able to modify a more interesting memory location. If attackers already control a pointer they could overwrite the target memory location directly instead of going through an extra level of indirection.

Our approach does not *detect* when a buffer overflow has occurred. It is, however, possible to easily and efficiently add such detection as an extension to dnmalloc. A technique similar to the one used in [142, 100] could be added to the allocator by placing a random number at the top of a chunk (where the old management information used to be) and by mirroring that number in the management information. Before performing any heap operation (i.e., malloc, free, coalesce, etc) on a chunk, the numbers would be compared and if changed, it could report the attempted exploitation of a buffer overflow. This of course only detects overflows which try to exploit the original problem that [142, 100] and we address: overwriting of the management information. If an overflow overwrites a pointer in another chunk and no heap operations are called, then the overflow will go undetected.

A major advantage of this approach over [142] is that it does not rely on a global secret value, but can use a per-chunk secret value. While this approach would improve detection of possible attacks, it does not constitute the underlying security principle, meaning that the security does not rely on keeping values in memory secret.

Finally, our countermeasure (as well as other existing ones [68, 142]) focuses on protecting this memory management information, it does not provide strong protection to pointers stored by the program itself in the heap. There are no efficient mechanisms yet to transparently and deterministically protect these pointers from modification through all possible kinds of heap-based buffer overflows. In order to achieve reasonable performance, countermeasure designers have focused on protecting the most targeted pointers. Extending the protection to more pointers without incurring a substantial performance penalty remains a challenging topic for future research.

## 4.6 Related work[14]

Many countermeasures for code injection attacks exist. In this section, we briefly describe the different approaches that could be applicable to protecting against heap-based buffer overflows, but will focus more on the countermeasures which are designed specifically to protect memory allocators from heap-based buffer overflows.

### 4.6.1 Protection from attacks on heap-based vulnerabilities

There are two types of allocators that try to detect or prevent heap overflow vulnerabilities: debugging allocators and runtime allocators. Debugging allocators are allocators that are meant to be used by the programmer. They can perform extra checks before using the management information stored in the chunks or

[14]This section discusses work closely related to our countermeasure, a more extensive discussion on the topic of countermeasures for code injection attacks can be found in section 2.4

ensure that the chunk is allocated in such a way that it will cause an error if it is overflowed or freed twice. Runtime allocators are meant to be used in final programs and try to protect memory allocators by performing lightweight checks to ensure that chunk information has not been modified by an attacker.

### Debugging memory allocators

*Dlmalloc* has a debugging mode that will detect modification of the memory management information. When run in debug mode the allocator will check to make sure that the next pointer of the previous chunk equals the current chunk and that the previous pointer of the next chunk equals the current chunk. To exploit a heap overflow or a double free vulnerability, the pointers to the previous chunk and the next chunk must be changed.

Electric fence [129] is a debugging library that will detect both underflows and overflows on heap-allocated memory. It operates by placing each chunk in a separate page and by either placing the chunk at the top of the page and placing a guard page before the chunk (underflow) or by placing the chunk at the end of the page and placing a guard page after the chunk (overflow). This is an effective debugging library but it is not realistic to use in a production environment because of the large amount of memory it uses (every chunk is at least as large as a page, which is 4kb on IA32) and because of the large performance overhead associated with creating a guard page for every chunk. To detect dangling pointer references, it can be set to never release memory back to the system. Instead, Electric fence will mark it as inaccessible, this will however result in an even higher memory overhead.

### Runtime allocators

Robertson et al. [142] designed a countermeasure that attempts to protect against attacks on the dlmalloc library management information. This is done by changing the layout of both allocated and unallocated memory chunks. To protect the management information a checksum and padding (as chunks must be of double word length) is added to every chunk. The checksum is a checksum of the management information encrypted (XOR) with a global read-only random value, to prevent attackers from generating their own checksum. When a chunk is allocated the checksum is added and when it is freed the checksum is verified. Thus if an attacker overwrites this management information with a buffer overflow a subsequent free of this chunk will abort the program because the checksum is invalid. However, this countermeasure can be bypassed if an information leak exists in the program that would allow the attacker to print out the encryption key. The attacker can then modify the chunk information and calculate the correct value of the checksum. The allocator would then be unable to detect that the chunk information has been changed by an attacker.

*Dlmalloc* 2.8.x also contains extra checks to prevent the allocator from writing into memory that lies below the heap (this however does not stop it from writing into memory that lies above the heap, such as the stack). It also offers a slightly modified version of the Robertson countermeasure as a compile-time option.

ContraPolice [100] also attempts to protect memory allocated on the heap from buffer overflows that would overwrite memory management information associated with a chunk of allocated memory. It uses the same technique as proposed by StackGuard [49], i.e., canaries, to protect these memory regions. It places a randomly generated canary both before and after the memory region that it protects. Before exiting from a string or memory copying function, a check is done to ensure that, if the destination region was on the heap, the canary stored before the region matches the canary stored after the region. If it does not, the program is aborted. While this does protect the contents of other chunks from being overwritten using one of these functions, it provides no protection for other buffer overflows. It also does not protect a buffer from overwriting a pointer stored in the same chunk. This countermeasure can also be bypassed if the canary value can be read: the attacker could write past the canary and make sure to replace the canary with the same value it held before.

Although no performance measurements were done by the author, it is reasonable to assume that the performance overhead would be fairly low.

Recent versions of glibc [68] have added an extra sanity check to its allocator: before removing a chunk from the doubly linked list of free chunks, the allocator checks if the backward pointer of the chunk that the unlinking chunk's forward pointer points to is equal to the unlinking chunk. The same is done for the forward pointer of the chunk's backward pointer. It also adds extra sanity checks that make it harder for an attacker to use the previously described technique of attacking the memory allocator. However, recently, several attacks on this countermeasure were published [130]. Although no data is available on the performance impact of adding these lightweight checks, it is reasonable to assume that no performance loss is incurred by performing them.

DieHard [15] in standalone mode is a memory allocator that will add protection against accidental overflows of buffers by randomizing allocations. The allocator will separate memory management information from the data in the heap. It will also try to protect the contents of a chunk by allocating chunks of a specific chunk size into a region at random positions in the region. This will make it harder for an application to accidentally overwrite the contents of a chunk, however a determined attacker could still exploit it by replicating the modified contents over the entire region. Performance for this countermeasure is very good for some programs (it improves performance for some) while relatively high for others. The work on DieHard was done in parallel to the countermeasure discussed in this document.

### 4.6.2 Alternative approaches

Other approaches that protect against the more general problem of buffer overflows also protect against heap-based buffer overflows. In this section, we give a brief overview of this work. A more extensive survey can be found in [198].

#### Safe languages

Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. Examples of such languages include Java and ML but these are not in the scope of our discussion. However there are safe languages [87, 74, 121, 105, 55, 98] that remain as close to C or C++ as possible, these are generally referred to as safe dialects of C. While some safe languages [44] try to stay more compatible with existing C programs, use of these languages may not always be practical for existing applications.

#### Compiler-based countermeasures

Bounds checking [8, 89, 144, 193] is the ideal solution for buffer overflows, however performing bounds checking in C can have a severe impact on performance or may cause existing object code to become incompatible with bounds checked object code.

Protection of all pointers as provided by PointGuard [48] is an efficient implementation of a countermeasure that will encrypt (using XOR) all pointers stored in memory with a randomly generated key and decrypts the pointer before loading it into a register. To protect the key, it is stored in a register upon generation and is never stored in memory. However attackers could guess the decryption key if they were able to view several different encrypted pointers. Another attack, described in [4] describes how an attacker could bypass PointGuard by overwriting a particular byte of the pointer. By modifying one byte, the pointer value has changed but the three remaining bytes will still decrypt correctly because of the weakness of XOR encryption. This significantly reduces the randomness (if only one byte needs to be overwritten, an attacker has a 1 in 256 chance of guessing the correct one, if two bytes are overwritten the chances are 1 in 65536, which is still significantly less than 1 in $2^{32}$.

Another countermeasure that protects all pointers is the Security Enforcement Tool [194] where runtime protection is performed by keeping a status bit for every byte in memory, that determines if writing to a specific memory region via an unsafe pointer is allowed or not.

### Operating system-based countermeasures

Non-executable memory [170, 161] tries to prevent code injection attacks by ensuring that the operating system does not allow execution of code that is not stored in the text segment of the program. This type of countermeasure can however be bypassed by a return-into-libc attack [188] where an attacker executes existing code (possibly with different parameters).

Randomized instruction sets [12, 93] also try to prevent an attacker from executing injected code by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. However, software based implementations of this countermeasure incur large performance costs, while a hardware implementation is not immediately practical. Determined attackers may also be able to guess the encryption key and, as such, be able to inject code [163].

Address randomization [170, 18] is a technique that attempts to provide security by modifying the locations of objects in memory for different runs of a program, however the randomization is limited in 32-bit systems (usually to 16 bits for the heap) and as a result may be inadequate for a determined attacker [151].

### Library-based countermeasures

LibsafePlus [9] protects programs from all types of buffer overflows that occur when using unsafe C library functions (e..g *strcpy*). It extracts the sizes of the buffers from the debugging information of a program and as such does not require a recompile of the program if the symbols are available. If the symbols are not available, it will fall back to less accurate bounds checking as provided by the original Libsafe [11] (but extended beyond the stack). The performance of the countermeasure ranges from acceptable for most benchmarks provided to very high for one specific program used in the benchmarks

### Execution monitoring

In this section we describe two countermeasures that monitor the execution of a program and prevent transferring control-flow which could be unsafe.

Program shepherding [96] is a technique that monitors the execution of a program and will disallow control-flow transfers[15] that are not considered safe. An example of a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter. As a result, the performance impact of this countermeasure is significant for some programs, but acceptable for others.

[15]Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. Performance overhead may be acceptable for some applications, but may be prohibitive for others.

## 4.7 Conclusion

In this paper we examined the security of several memory allocators. We discussed how they could be exploited and showed that most memory allocators are vulnerable to code injection attacks.

Afterwards, we presented a redesign for existing memory allocators that is more resilient to these attacks than existing allocator implementations. We implemented this design by modifying an existing memory allocator. This implementation has been made publicly available. We demonstrated that it has a negligible, sometimes even beneficial, impact on performance. The overhead in terms of memory usage is very acceptable. Although our approach is straightforward, surprisingly, it offers stronger security than comparable countermeasures with similar performance overhead because it does not rely on the secrecy of random numbers stored in memory.

# Chapter 5

# Conclusion

Using the survey presented in chapter 2 we identified a number of areas in which countermeasures could be designed that improve the state-of-the-art of current countermeasures. The main design premise was to model countermeasures the same way that code and data are generally stored in separate memory regions in modern operating systems. This same separation can be applied to data: regular data can be stored separately from code data. If the mechanisms that are used to implement the abstractions provided by high level-languages are stored separately from regular data, it becomes harder for attackers to exploit them to gain control of the program's execution flow.

In this chapter, we summarize our main contributions and present a number of countermeasures that are currently being designed and implemented. We also discuss future research opportunities, challenges and application domains that new technologies offer when it comes to building countermeasures.

## 5.1 Contributions

During the four years of research that led to this dissertation, a number of contributions were made to the systems security domain.

First, an extensive survey was performed of vulnerabilities and countermeasures that are published for C and C++ [197, 198]. This survey presents a classification and evaluation framework for existing and future countermeasures, allowing countermeasures to be more easily compared with respect to several important criteria like efficiency, completeness and automatization.

Based on this survey, a more structured approach to designing vulnerabilities was suggested [199]. This approach allows a countermeasure designer to more easily consider the impact of a specific countermeasure on the entire system and allows an evaluation of the possible benefits and drawbacks without first doing an expensive implementation.

This methodology formed the basis in the design of three countermeasures for buffer overflows in different segments of memory, based on the premise that code data and regular data should be separated from each other.

- A countermeasure was designed and implemented which better protects against stack-based buffer overflows (see Chapter 3). This countermeasure provides protection by dividing the stack into multiple stacks, depending on the type of data stored on the stack. Each type of data is assigned a target and source value. The target value depends on how valuable the type of data is to an attacker as a target for attack. For example the return address has a high target value because control over this memory location would give attackers immediate control of the execution flow, while an array of characters has a low target value, because being able to modify the contents of an array of characters generally does not allow an attacker to gain control of the program's execution flow. The source value depends on how likely an attacker can use the type of data to perform an attack. An array of characters has a high source value, because arrays of characters are generally the ones that are vulnerable to buffer overflows since they are often used with functions like *strcpy*. A return address has a low source value, because an attacker generally does not have direct control of this data without using a vulnerability. Using these values, the stack is divided into multiple stacks: data with low source and high target values are stored together and vice versa. Depending on the amount of memory that is available compared to the amount of protection that is required, the number of stacks can be increased or reduced. Unlike similar countermeasures that are equally efficient and automatic, this countermeasure does not rely on the use of random numbers that must remain secret. This means that the countermeasure retains its security properties even if attackers have read access to memory.

- In addition, a countermeasure was designed and implemented which provides better protection against heap-based buffer overflows (see Chapter 4). This countermeasure consists of a technique to efficiently separate the memory management information that is often a target for attack from the regular data stored on the heap. By storing the management information in a separate contiguous memory region, we can prevent an attacker from overwriting it directly using a heap-based buffer overflow. Our implementation demonstrates that this can be done efficiently. Like the stack-based countermeasure, it offers better protection than countermeasures that offered similar performance results because it does not rely on the use of random numbers.

- A third countermeasure that was designed provides similar protection against exploitation of buffer overflows in the data and bss segments [202]. This countermeasure was not implemented because the technique was similar to the one used in the stack-based countermeasure. In this countermeasure, data is also separated from other data based on similar target and source values.

These countermeasures improve the protection provided against buffer overflows while retaining the efficiency and automatization that similar countermeasures that focus on these two properties offer.

## 5.2 Future work

The author of the dissertation is currently involved in the design and implementation of a number of new countermeasures that can provide better protection against the vulnerabilities discussed in this dissertation. We discuss two countermeasures here, while a third is discussed in the next section.

### 5.2.1 A bounds checker for pointer arithmetic

During a research stay at Stony Brook University, the author designed a third countermeasure that performs bounds checking for C. This is achieved by introducing extra checks when pointer arithmetic is performed. Current bounds checkers usually perform their checks when a pointer is dereferenced. By doing the check at calculation-time instead of when the pointer is dereferenced, a performance increase should be achieved. Whenever an object is allocated in memory, the entire memory space of that object is marked with a unique value, which we call a label. Whenever pointer arithmetic is performed with a pointer that refers to an object, the label of the object the pointer refers to is compared to the label of the result of the arithmetic. If they are not equal, then a pointer has been created that points out of bounds of the object. This work is still in progress, however, we expect the countermeasure to be complete and compatible with existing code. Our focus has been mainly on making the bounds checker more efficient and on ensuring its scalability, so that it could be used in production systems.

### 5.2.2 A countermeasure for dangling pointer references

This countermeasure is based on the experience that was gained from developing the bounds checker, but focuses on dangling pointer references. Existing countermeasures for dangling pointer references [53] suffer from a high performance impact because many checks must be performed to prevent dangling pointers from being used. The countermeasure that will be developed is expected to be more efficient by keeping a reverse mapping from the object to the pointer. When a pointer is set to refer to an object, a reverse mapping from the object to the pointer is added. When an object is freed, the mapping is used to invalidate all pointers that refer to this object. When the program tries to dereference such a dangling pointer, a crash will occur, preventing the pointer from being used in an invalid manner.

## 5.3 Future research opportunities and application domains

In this section, we discuss research opportunities offered and challenges posed by a number of new and recent technologies with respect to the design of new countermeasures. We also discuss possible new application domains for countermeasures against code injection attacks.

### 5.3.1 Embedded systems and mobile devices

New technologies like embedded systems, sensor networks and mobile devices present new challenges. These systems must often work with very limited resources, making the use of C a necessity, resulting in the same problems as in traditional architectures. Due to the specific requirements of these systems, the countermeasures will also look different from the ones for traditional systems, although the same basic ideas could be ported to these devices. Many types of embedded devices exist, running on a large number of different architectures. While the large amount of architectures makes it harder for attackers to exploit vulnerabilities, it also makes it harder to build countermeasures.

In August 2006, a number of vulnerabilities [125, 126] were discovered in LibTIFF [1]. LibTIFF is used in a number of desktop operating systems, like Linux and Mac OS X. It is also used on the Apple iPhone, where this vulnerability was widely exploited by users of the iPhone [119] to perform a “jailbreak” [2]. This vulnerability can be triggered in both MobileMail (the iPhone mail client) and MobileSafari (the iPhone web browser) and as a result is remotely exploitable by letting the user browse to a site containing a specific TIFF file or by emailing a TIFF file to the user.

This vulnerability was also present on another mobile device: the Sony PlayStation Portable, where it was also exploited to allow behavior not condoned by the manufacturer. In this case, the vulnerability was used to gain more permissions to allow users to run homebrew[3] games.

[1]LibTIFF is a library for reading and writing TIFF files, a popular image format

[2]By default, it is not possible for users to install additional native applications on the iPhone. The term jailbreaking refers to the escaping of these limitations, allowing users to gain full control of the device.

[3]Homebrew games are games that are typically produced by consumers and are generally not authorized (or digitally signed) by the manufacturer of the product, resulting in the need to circumvent security restrictions on the device before they can be played.

These examples show that software originally designed for desktop environments is being ported widely to these new devices, resulting in the same types of vulnerabilities being present in these devices. As more and more of these devices enter the market, similar vulnerabilities will be discovered and exploited.

Very few countermeasures currently exist for embedded systems. The most important countermeasure that is currently deployed on these systems is the stack cookie protection in Windows CE 6. This countermeasure is based on the StackGuard countermeasure [49], which places a random (secret) number before the return address upon function entry and verifies that the number has not changed before returning from the function. The underlying idea is that an attacker will be unable to overwrite the return address without also modifying the random number. This number must remain a secret, otherwise an attacker can just restore the number while overwriting. This is an inherent weakness in this approach.

Many of the countermeasures that are currently in use on desktop systems can be ported to embedded devices. However, due to limited memory and processing power, efficiency becomes a more important concern on these devices. Several other limitations in the architecture may also be an issue when trying to port countermeasures to these architectures. For example, many architectures have no support for paging, making it hard to implement a countermeasure like address space layout randomization (ASLR) on these devices.

Another important issue with porting desktop countermeasures to embedded devices is the fact that many desktop countermeasures were designed with a specific architecture or operating system in mind. This can make porting more difficult and prone to being bypassed. When a countermeasure is ported, the countermeasure developer must ensure that the countermeasure cannot easily be bypassed on the new platform. An example of such porting going wrong occurred when Microsoft ported the StackGuard countermeasure [49] from GCC to Visual Studio [31]. The specifics of the Windows operating system were not taken into account, which resulted in attackers being able to bypass the countermeasure by ignoring the return address and continuing to write on the stack until they overwrote the function pointers used for exception handling. Subsequently, an exception would be generated by the attackers and their injected code would be executed [113]. A possible way to prevent these types of problems when porting countermeasures to a new platform is to use machine model aided countermeasures [199].

### 5.3.2 Virtual machine monitors

Other technologies like virtual machine monitors can also form the basis for designing new countermeasures. Due to the rise of efficient virtual machine monitors, it is possible to design countermeasures at a lower level than was previously possible: if a program is running in a virtual machine, it is possible to modify the entire architecture to fix a security problem, rather than having to work around specific problems of a certain architecture. This can be useful in designing prototypes that can either be applied to future revisions of the architecture or can be applied in an environment in which virtual machine monitors are used.

We are currently in the process of designing and implementing a countermeasure that makes use of a virtual machine monitor to protect against the vulnerabilities discussed in this dissertation. The countermeasure is based on the observation that many exploits that result in code execution, rely on modifying the value of a pointer. In this countermeasure, all pointers are stored in read-only memory preventing them from being modified directly by a vulnerability. Whenever the program writes to a pointer, the compiler generates a special instruction that can be used to write to this pointer. This instruction will be emulated by the virtual machine monitor. If a vulnerability exists in a program that allows an attacker to overwrite arbitrary memory locations, this is prevented because only the special instruction can write to pointers. Since attackers are not be able to generate this special instruction until they are able to achieve code execution, this countermeasure would provide a significant improvement in protection against exploitation of these vulnerabilities. Once the prototype has determined whether this approach is feasible or not, it could be used as the basis for possible later revisions of the architecture.

# Bibliography

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., November 2005. ACM.

[2] Robert P. Abbott, Janet S. Chin, James. E. Donnelley, William L. Konigsford, Shigeru Tokubo, and Douglas A. Webb. Security analysis and enhancements of computer operating systems. Technical report, 1976.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[4] Steven Alexander. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine*, 30(3), June 2005.

[5] anonymous. Once upon a free(). *Phrack*, 57, 2001.

[6] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, Berkeley, California, USA, May 2002. IEEE Computer Society, IEEE Press.

[7] Taimur Aslam. A taxonomy of security faults in the unix operating system. Master's thesis, Purdue University, 1995.

[8] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, U.S.A., June 1994. ACM.

[9] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proc. of the 13th USENIX Security Symp.*, San Diego, CA, August 2004.

[10] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows: Research articles. *Software – Practice & Experience*, 36(9):971–998, 2006.

[11] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX 2000 Annual Technical Conference Proceedings*, pages 251–262, San Diego, California, U.S.A., June 2000. USENIX Association.

[12] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, District of Columbia, U.S.A., October 2003. ACM.

[13] BBP. BSD heap smashing. `http://www.security-protocols.com/modules.php?name=News&file=article&sid=1586`, May 2003.

[14] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168, Ottawa, Ontario, Canada, 2006. ACM Press.

[15] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, Ottawa, Canada, June 2006.

[16] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, Snowbird, Utah, U.S.A., June 2001.

[17] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 1–12, Seattle, Washington, U.S.A., November 2002. ACM, ACM Press.

[18] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

[19] Sandeep Bhatkar and R. Sekar. Data space randomization. Technical report, Secure Systems Labratory, Stony Brook University, February 2008.

[20] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005. USENIX Association.

[21] Richard Bisbey II and Dennis Hollingsworth. Protection analysis project: Final report. Technical report, Information Sciences Institute, University of Southern California, 1978.

[22] Matt Bishop. A taxonomy of UNIX system and network vulnerabilities. Technical Report CSE-9510, Department of Computer Science, University of California at Davis, May 1995.

[23] Matt Bishop. Vulnerability analysis. In *Proceedings of Recent Advances in Intrusion Detection 1999*, pages 125–136, West Lafayette, Indiana, U.S.A., September 1999.

[24] Matt Bishop and Dave Bailey. A critical analysis of vulnerability taxonomies. Technical report, Department of Computer Science, University of California at Davis, 1996.

[25] Bruno Blanchet, Patrick Cousot, Radhia Cousot, J&#233;rome Feret, Laurent Mauborgne, Antoine Min&#233;, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, San Diego, California, USA, 2003. ACM Press.

[26] blexim. Basic integer overflows. *Phrack*, 60, December 2002.

[27] Hans Boehm. Conservative gc algroithmic overview. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html`.

[28] Hans Boehm. A garbage collector for c and c++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[29] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software, Practice and Experience*, 18(9):807–820, September 1988.

[30] Brandon Bray. Compiler security checks in depth. `http://msdn.microsoft.com/library/en-us/dv\_vstechart/html/vctchCompilerSecurityChecksInDepth.asp`, February 2002.

[31] Brandon Bray. Security improvements to the whidbey compiler. `http://weblogs.asp.net/branbray/archive/2003/11/11/51012.aspx`, November 2003.

[32] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium*, San Diego, California , U.S.A., March 2007. Internet Society.

[33] Danilo Bruschi, Emilia Rosti, and Banfi R. A tool for pro-active defense against the buffer overrun attack. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS)*, volume 1485 of *Lecture Notes in Computer Science*, pages 17–31, Louvain-la-Neuve, Belgium, 1998. Springer.

[34] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[35] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, June 2000. ISSN: 0038-0644.

[36] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, U.S.A., November 2006. ACM Press.

[37] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 147–160, Seattle, Washingotn, U.S.A., November 2006. Usenix.

[38] Shun Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005. IEEE Press.

[39] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of the 14th USENIX Security Symp.*, Baltimore, MD, August 2005.

[40] Brian V. Chess. Improving Computer Security using Extended Static Checking. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 160–173, Berkeley, California, U.S.A., May 2002. IEEE Press.

[41] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[42] T. Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed*

*Computing Systems*, pages 409–420, Phoenix, Arizona, USA, April 2001. IEEE Computer Society, IEEE Press.

[43] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, Braga, Portugal, March 2007. Springer-Verlag.

[44] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, U.S.A., 2003. ACM.

[45] Matt Conover. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, 1999.

[46] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–200, Washington, District of Columbia, U.S.A., August 2001. USENIX Association.

[47] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Eric Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Proceedings of Linux Expo 1999*, Raleigh, North Carolina, U.S.A., May 1999.

[48] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

[49] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[50] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference & Exposition*, volume 2, pages 119–129, Hilton Head, South Carolina, U.S.A., January 2000.

[51] Joan Daemen and Vincent Rijmen. Rijndael, the advanced encryption standard. *Dr. Dobb's Journal*, 26(3), March 2001.

[52] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceeding of the 28th international conference on Software engineering*, pages 162–171, Shanghai, China, 2006. ACM Press.

[53] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, Philadelphia, Pennsylvania, U.S.A., 2006. IEEE Computer Society.

[54] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, Ottawa, Ontario, Canada, 2006. ACM Press.

[55] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 69–80, San Diego, California, U.S.A., June 2003. ACM.

[56] Igor Dobrovitski. Exploit for CVS double free() for linux pserver. `http://seclists.org/lists/bugtraq/2003/Feb/0042.html`, February 2003.

[57] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulation in C programs via integer analysis. In *Proceedings of the Eight International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212, Paris, France, July 2001. Springer-Verlag.

[58] Computer Economics. 2001 economic impact of malicious code attacks. `http://www.computereconomics.com/article.cfm?id=133`, January 2002.

[59] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[60] Úlfar Erlingsson. Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research, November 2007.

[61] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM.

[62] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[63] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, U.S.A., May 1999. IEEE Computer Society, IEEE Press.

[64] Christof Fetzer and Zhen Xiao. Detecting heap smashing attacks through fault containment wrappers. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, pages 80–89, New Orleans, Lousiana, U.S.A., October 2001. IEEE Computer Society, IEEE Press.

[65] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997.

[66] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, California, U.S.A., May 1996. IEEE Computer Society, IEEE Press.

[67] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, Washington, District of Columbia, U.S.A., August 2001. USENIX Association.

[68] Free Software Foundation. The gnu c library. `http://www.gnu.org/software/libc`.

[69] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 345–354, Washington, District of Columbia, U.S.A., October 2003. ACM Press.

[70] Anup K. Ghosh and Tom O'Connor. Analyzing programs for vulnerability to buffer overrun attacks. In *Proceedings of the 21st NIST-NCSC National Information Systems Security Conference*, pages 274–382, October 1998.

[71] Wolfram Gloger. ptmalloc. `http://www.malloc.de/en/`.

[72] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, California, U.S.A., July 1996. USENIX Association.

[73] Richard Grimes. Preventing buffer overflows in C++. *Dr Dobb's Journal: Software Tools for the Professional Programmer*, 29(1):49–52, January 2004.

[74] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[75] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, New York, New York, U.S.A., June 1993.

[76] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th international conference on Software engineering*, pages 232–241, Shanghai, China, 2006. ACM Press.

[77] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136, San Francisco, California, U.S.A., January 1992. USENIX Association.

[78] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03)*, San Diego, California, U.S.A., February 2003. Internet Society.

[79] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[80] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, pages 73–84, Vancouver, BC, Canada, 2004. ACM Press.

[81] Gerard J. Holzmann. Static source code checking for user-defined properties. In *Proceedings of The 6th World Conference on Integrated Design & Process Technology*, Pasadena, California, U.S.A., 2002. Society for Design and Process Science.

[82] John D. Howard. *An Analysis Of Security Incidents On The Internet 1989-1995.* PhD thesis, Carnegie Mellon University., 1997.

[83] Michael Howard and David LeBlanc. *Writing Secure Code.* Microsoft Press, 2001.

[84] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 2–12, Ottawa, Ontario, Canada, 2006. ACM Press.

[85] Cisco Systems Inc. Cisco 2007 annual security report. `http://www.cisco.com/web/about/security/cspo/docsCisco2007Annual_Security_Report.pdf`.

[86] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001. Order Nr 245470.

[87] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[88] 'Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the 1st ACM SIGPLAN International Symposium on Memory Management*, pages 26–36, Vancouver, British Columbia, Canada, October 1998. ACM, ACM Press.

[89] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

[90] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999.

[91] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001.

[92] Michel MaXX Kaempf. Sudo ¡ 1.6.3p7-2 exploit. `http://packetstormsecurity.org/0211-exploits/hudo.c`.

[93] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, District of Columbia, U.S.A., October 2003. ACM.

[94] Samuel C. Kendall. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer 1983 Conference*, pages 5–16, Toronto, Ontario, Canada, July 1983. USENIX Association.

[95] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall Software Series, second edition edition, 1988.

[96] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 2002. USENIX Association.

[97] klog. The frame pointer overwrite. *Phrack*, 55, 1999.

[98] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[99] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley & Sons, March 2004.

[100] Andreas Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks. `http://www.synflood.at/contrapolice/`, November 2003.

[101] Lap-chung Lam and Tzi-cker Chiueh. Checking array bound violation using segmentation hardware. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 388–397, Yokohama, Japan, 2005. IEEE Computer Society.

[102] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. Technical report, 1993.

[103] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, Washington, District of Columbia, U.S.A., August 2001. USENIX Association.

[104] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

[105] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[106] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chigago, Illinois, June 2005. ACM Press.

[107] Doug Lea and Wolfram Gloger. malloc-2.7.2.c. Comments in source code.

[108] Doug Lea and Wolfram Gloger. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[109] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag, 2003.

[110] Kyung-Suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, San Francisco, California, U.S.A., August 2002. USENIX Association.

[111] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, April 2003.

[112] Cullen Lin, Mohan Rajagopalan, Scott Baker, Christian Collberg, Saumya Debray, and John Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, U.S.A., August 2005. USENIX Association.

[113] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. `http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf`, September 2003.

[114] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, British Columbia, Canada, August 2006. USENIX Association.

[115] Matt Messier and John Viega. Safe C string library V1.0.2. `http://www.zork.org/safestr`, November 2003.

[116] Microsoft. Vault: a programming language for reliable systems. `http://research.microsoft.com/vault/`.

[117] Microsoft. Buffer overrun in RPC interface could allow code execution. `http://www.microsoft.com/technet/security/bulletin/MS03-026.asp`, July 2003.

[118] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – consistent, safe string copy and concatenation. In *Proceedings of the 1999 USENIX Annual Technical Conference (FREENIX Track)*, pages 175–178, Monterey, California, U.S.A., June 1999. USENIX Association.

[119] HD Moore. Cracking the iphone. `http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html`.

[120] National Institute of Standards and Technology. National vulnerability database statistics. `http://nvd.nist.gov/statistics.cfm`.

[121] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., January 2002. ACM.

[122] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking Entire Programs without Recompiling. In *Informal Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004.

[123] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2005. Internet Society.

[124] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *Proceedings of International Symposium on Software Security 2002*, pages 133–153, Tokyo, Japan, November 2002.

[125] Tavis Ormandy. Libtiff next rle decoder remote heap buffer overflow vulnerability. `http://www.securityfocus.com/bid/19282`, Aug 2006.

[126] Tavis Ormandy. Libtiff tifffetchshortpair remote buffer overflow vulnerability. `http://www.securityfocus.com/bid/19283`, Aug 2006.

[127] Hilmi Özdoğanoğlu, T. N. Vijaykumar, Carla E. Brodley, Ankit Jalote, and Benjamin A. Kuperman. SmashGuard: A hardware solution to prevent security attacks on the function return address. Technical Report TR-ECE 03-13, Purdue University, February 2004.

[128] Harish Patil and Charles N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience*, 27(1):87–110, January 1997.

[129] Bruce Perens. Electric fence 2.0.5. `http://perens.com/FreeSoftware/`.

[130] Phantsmal Phantasmagoria. The malloc maleficarum. `http://lists.grok.org.uk/pipermail/full-disclosure/2005-October/037905.html`.

[131] Frank Piessens. A taxonomy of causes of software vulnerabilities in internet software. In *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 47–52, Annapolis, Maryland, U.S.A., November 2002. IEEE Computer Society, IEEE Press.

[132] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, Seattle, Washington, U.S.A., 2005. Springer.

[133] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, San Antonio, Texas, U.S.A., June 2003. USENIX Association.

[134] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX Track)*, Boston, Massachusetts, U.S.A., June 2001. USENIX Association.

[135] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

[136] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[137] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation*, San Francisco, California, USA, December 2004. USENIX Association.

[138] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 82–90, Tuscon, Arizona,U.S.A., December 2004. IEEE Press.

[139] Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 354–363, Alexandria, VA, USA, 2005. ACM Press.

[140] rix. Smashing C++ VPTRs. *Phrack*, 56, 2000.

[141] Tim Robbins. Libformat. `http://www.securityfocus.com/tools/1818`, October 2001.

[142] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., October 2003. USENIX Association.

[143] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, British Columbia, Canada, June 2000. ACM.

[144] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2004. Internet Society.

[145] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[146] Fred B. Schneider. Least privilege and more. *IEEE Security & Privacy*, 1(5):55–59, September/October 2003.

[147] scut. Exploiting format string vulnerabilities. `http://www.team-teso.net/articles/formatstring/`, 2001.

[148] Secure Software, Inc. RATS website. `http://www.securesw.com/download\_rats.htm`.

[149] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, California, U.S.A., May 2001. IEEE Computer Society, IEEE Press.

[150] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, Washington, District of Columbia, U.S.A., October 2007. ACM, ACM Press.

[151] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, District of Columbia, U.S.A., October 2004. ACM, ACM Press.

[152] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–218, Washington, District of Columbia, U.S.A., August 2001. USENIX Association.

[153] Zili Shao, Qingfeng Zhuge, Yi He, and Edwin H. M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, Nevada, U.S.A., December 2003. IEEE Press.

[154] Axel Simon and Andy King. Analyzing string buffers in C. In H. Kirchner and C. Ringeissen, editors, *International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer, September 2002.

[155] Christopher Small. A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 175–184, Portland, Oregon, U.S.A., June 1997. USENIX Association.

[156] Alexey Smirnov and Tzi-cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2005. Internet Society.

[157] Nathan P. Smith. Stack smashing vulnerabilities in the unix operating system. `http://reality.sgi.com/nate/machines/security/nate-buffer.ps`, 1997.

[158] Alexander Snarskii. Libparanoia. `http://www.lexa.ru/snar/libparanoia/`.

[159] Alexander Snarskii. FreeBSD libc stack integrity patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, February 1997.

[160] Brian D. Snow. The future is not assured – but it should be. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 240–241, Oakland, California, U.S.A., May 1999. IEEE Computer Society, IEEE Press.

[161] Solar Designer. Non-executable stack patch. `http://www.openwall.com`.

[162] Solar Designer. JPEG COM marker processing vulnerability in netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt`, July 2000.

[163] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, U.S.A., August 2005. Usenix.

[164] Eugene H. Spafford. Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[165] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, April 1992. ISSN: 0038-0644.

[166] W. Richard Stevens. *Advanced Programming in the UNIX enironment*. Addison-Wesley, 1993.

[167] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition edition, 1997.

[168] Andrew Suffield. Bounds checking for C and C++. Technical report, Imperial College, 2003.

[169] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, Boston, MA, USA, 2004. ACM Press.

[170] The PaX Team. Documentation for the PaX project. `http://pageexec.virtualave.net/docs/`.

[171] Ruud van der Pas. Memory hierarchy in cache-based systems. Technical Report 817-0742-10, Sun Microsystems, Sant a Clara, California, U.S.A., November 2002.

[172] Vendicator. Documentation for stack shield. `http://www.angelfire.com/sk/stackshield`.

[173] Vendicator. Documentation for stackshield. `http://www.angelfire.com/sk/stackshield`.

[174] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, U.S.A., December 2000.

[175] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[176] Kevin Poulson vs. U.S Customs and Border Protection. Declaration of shari suzuki in opposition to motion for summary judgement. `http://wiredblogs.tripod.com/27BStroke6/suzukidecl.pdf`, May 2006.

[177] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, Oakland, California, U.S.A., May 2001. IEEE Computer Society, IEEE Press.

[178] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Networking and Distributed System Security Symposium 2000*, pages 3–17, San Diego, California, U.S.A., February 2000.

[179] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, Asheville, North Carolina, U.S.A., December 1993. ACM.

[180] Yoav Weiss and Elena Gabriela Barrantes. Known/chosen key attacks against software instruction set randomization. In *22nd Annual Computer Security Applications Conference*, Miami Beach, Florida, U.S.A., December 2006. IEEE Press.

[181] David A. Wheeler. Flawfinder website. `http://www.dwheeler.com/flawfinder/`.

[182] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*, 3.010 edition, March 2003.

[183] Wikipedia. Project triangle. `http://en.wikipedia.org/wiki/Project_triangle`.

[184] Wikipedia. Wikipedia entry for code injection. `http://en.wikipedia.org/wiki/Code_injection`.

[185] Wikipedia. Wikipedia entry for the sasser worm. `http://en.wikipedia.org/wiki/Sasser`.

[186] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of NORDSEC 2002: the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.

[187] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2003. Internet Society.

[188] Rafal Wojtczuk. Defeating Solar Designer's Non-executable Stack Patch. `http://www.insecure.org/sploits/non-executable.stack.problems.html`, 1998.

[189] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference, held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 327–336, Helsinki, Finland, 2003. ACM Press.

[190] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, October 2003. IEEE Computer Society, IEEE Press.

[191] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and K. Iyer Ravishankar. Architecture support for defending against buffer overflow attacks. In *Second Workshop on Evaluating and Architecting System dependabilitY*, pages 55–62, San Jose, California, U.S.A., October 2002.

[192] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, British Columbia, Canada, August 2006. USENIX Association.

[193] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, California, U.S.A., October-November 2004. ACM, ACM Press.

[194] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software*

*engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 307–316. ACM, ACM Press, September 2003.

[195] Yves Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003.

[196] Yves Younan. Dnmalloc 1.0. `http://www.fort-knox.org`, 2005.

[197] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++: a survey of vulnerabilities and countermeasures. *ACM Computing Surveys*. Submitted.

[198] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[199] Yves Younan, Wouter Joosen, and Frank Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005)*, College Park, Maryland, U.S.A., March 2005. IEEE, IEEE Press.

[200] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, Raleigh, North Carolina, U.S.A., December 2006.

[201] Yves Younan, Wouter Joosen, and Frank Piessens. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the Twenty-Second Annual Computer Security Applications Conference*, Miami, Florida, U.S.A., December 2006. IEEE Press.

[202] Yves Younan, Wouter Joosen, and Frank Piessens. Protecting global and static variables from buffer overflow attacks without overhead. Technical Report CW463, Departement Computerwetenschappen, Katholieke Universiteit Leuven, October 2006.

[203] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Technical Report CW419, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2005.

[204] Zen-parse. Wu-ftpd 2.6.1 exploit. `http://www.derkeiler.com/Mailing-Lists/securityfocus/vuln-dev/2001-12/0160.html`.

[205] Ge Zhu and Akhilesh Tyagi. Protection against indirect overflow attacks on pointers. In *Proceedings of the 2nd IEEE International Information Assurance Workshop (IWIA)*, pages 97–106, Charlotte, North Carolina, U.S.A., April 2004. IEEE Press.

[206] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 138–147, Washington, District of Columbia, U.S.A., November 2002. ACM.

# List of Publications

## Articles in international reviewed journals

- Y. Younan, W. Joosen and F. Piessens, Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures. ACM Computing Surveys. Submitted, January 2005. Conditionally accepted (minor revision), March 2007, Revised, May 2007.

- Y. Younan, W. Joosen, F. Piessens and H. Van den Eynden, Security of memory allocators for C and C++, Journal of Computer Security. Submitted, January 2007, Revised March 2008.

## Contributions at international conferences, published in proceedings

- Y. Younan, D. Pozza, F. Piessens, and W. Joosen, Extended protection against stack smashing attacks without performance loss, Twenty-Second Annual Computer Security Applications Conference, pp. 429-438, 2006

- Y. Younan, W. Joosen, and F. Piessens, Efficient protection against heap-based buffer overflows without resorting to magic, Information and Communications Security: 8th International Conference (Ning, P. and Qing, S. and Li, N., eds.), vol 4307, Lecture Notes in Computer Science, pp. 379-398, 2006

- Y. Younan, W. Joosen, and F. Piessens, Applying machinemodel-aided countermeasure design to improve memory allocator security, 22nd Chaos Communication Congress Proceedings (-, ed.), pp. 249-254, 2005

- Y. Younan, W. Joosen, and F. Piessens, A methodology for designing countermeasures against current and future code injection attacks, Proceedings of the Third IEEE International Information Assurance Workshop (Cole, J. and Wolthusen, S., eds.), pp. 3-20, 2005

## Contributions at international conferences, not published or only as abstract

- Y. Younan, W. Joosen, and F. Piessens, Applying machinemodel-aided countermeasure design to improve security, 5th System Administration and Network Engineering Conference, SANE, Delft, The Netherlands, May 15-19, 2006,

- Y. Younan, W. Joosen, and F. Piessens, Applying machinemodel-aided countermeasure design to improve security, 15th USENIX Security Symposium, Vancouver, Canada, August 1-4, 2006, Y. Younan, A methodological approach to designing protection against code injection attacks, 1st EuroSys Doctoral Workshop, Brighton, United Kingdom, October 23, 2005,

## Contributions at other conferences, not published or only as abstract

- Y. Younan, Security of memory allocators for C and C++, What The Hack, WTH, Liempde, The Netherlands, July 28-31, 2005, unpublished

- Y. Younan, Security of memory allocators for C and C++, ph-neutral, Berlin, Germany, May 27-29, 2005, unpublished

## Technical reports

- Y. Younan, F. Piessens, and W. Joosen, Protecting global and static variables from buffer overflow attacks without overhead, K.U.Leuven, Department of Computer Science, Report CW 463, October, 2006

- Y. Younan, W. Joosen, F. Piessens, and H. Van den Eynden, Security of memory allocators for C and C++, Department of Computer Science, K.U.Leuven, Report CW 419, July, 2005

- Y. Younan, W. Joosen, and F. Piessens, Code injection in C and C++: a survey of vulnerabilities and countermeasures, K.U.Leuven, Department of Computer Science, Report CW 386, July, 2004

# Biography

Yves Younan was born on January 16th, 1979 in Lachine, Canada. He received a Licentiate in Computer Science (the Belgian equivalent of a Master of Science in Computer Science) from the Vrije Universiteit Brussel. He graduated magna cum laude in September 2003 with the thesis "An overview of common programming security vulnerabilities and possible solutions" which was promoted by Prof. dr. Dirk Vermeir. In February 2004, he joined the DistriNet research group of the Department of Computer Science of the Katholieke Universiteit Leuven as a PhD student. During the course of his PhD studies, he was a visiting scholar from July 2007 until January 2008 at the Secure Systems Lab of the Department of Computer Science at Stony Brook University, where he collaborated with Prof. dr. R. Sekar.

# Dutch Summary

# Efficiënte tegenmaatregelen voor softwarekwetsbaarheden veroorzaakt door geheugenbeheerfouten

**Samenvatting**

Ondanks vele jaren van onderzoek en grote investeringen door bedrijven, is de ontwikkeling van veilige software nog steeds een groot probleem. Dit blijkt uit de gestage toename van de kwetsbaarheden die jaarlijks zijn gemeld. Snelle verspreidende wormen zoals de worm Code Red, die naar schatting een wereldwijd economisch verlies van $2,62 miljard heeft veroorzaakt, zullen vaak fouten in programma's uitbuiten om zich snel te verspreiden.

Kwetsbaarheden die kunnen uitgebuit worden door aanvallers voor het uitvoeren van code injectie aanvallen zijn een belangrijke vorm van implementatiefouten. De worm Code Red buit een bufferoverloop uit om willekeurige code te kunnen uitvoeren op de kwetsbare machine, waardoor hij zichzelf kan verspreiden door zich te kopiëren naar machines die hij besmet. Het wijdverspreide gebruik van C-achtige talen waar dergelijke kwetsbaarheden een belangrijk probleem zijn heeft het probleem verergerd.

In dit proefschrift onderzoeken we een aantal kwetsbaarheden in C-achtige talen, die door aanvallers kunnen worden uitgebuit voor het uitvoeren van code injectie aanvallen en bespreken we tegenmaatregelen die bescherming bieden tegen dit soort aanvallen. Dit proefschrift bestaat uit drie belangrijke onderdelen: het begint met de presentatie van een uitgebreide inventarisatie van de huidig bekende kwetsbaarheden en tegenmaatregelen, dit wordt gevolgd door een discussie van twee nieuwe tegenmaatregelen die gericht zijn op een betere bescherming tegen aanvallen op verschillende kwetsbaarheden terwijl die slechts een te verwaarlozen invloed hebben op performantie.

De inventarisatie biedt een uitgebreid en gestructureerd overzicht van kwetsbaarheden en tegenmaatregelen voor code injectie in C-achtige talen. Diverse tegenmaatregelen maken verschillende afwegingen in termen van performantie, effectiviteit, geheugenverbruik, compatibiliteit, enz. Dit maakt het moeilijk te beoordelen en vergelijken van de geschiktheid van de voorgestelde tegenmaatregelen in een gegeven context. Deze inventaristatie is een classificatie en evaluatie kader, op basis waarvan de voordelen en nadelen van tegenmaatregelen kunnen worden beoordeeld. Op basis van de opmerkingen en de conclusies die werden getrokken

uit de inventarisatie, zijn twee tegenmaatregelen ontworpen, geïmplementeerd en geëvalueerd.

De eerste tegenmaatregel die we beschrijven is een efficiënte tegenmaatregel tegen stapelvermorzelingsaanvallen. Onze tegenmaatregel maakt geen gebruik van geheime waarden (zoals kanaries) en beschermt tegen aanvallen waartegen gelijkaardige tegenmaatregelen niet beschermen. Onze techniek splitst de standaard stapel in meerdere stapels. De verdeling van de soorten gegevens aan één van de stacks is gebaseerd op de kans dat een specifiek data-element ofwel een doelwit ofwel een bron van aanvallen is. We hebben deze tegenmaatregel geïmplementeerd in een C-compiler voor Linux. De evaluatie toont aan dat de impact op performantie door het gebruik van onze tegenmaatregel verwaarloosbaar is.

De tweede tegenmaatregel beschermt tegen aanvallen op hoop-gebaseerde bufferoverlopen en zwevende wijzers. Het wijzigen van de beheersinformatie die gebruikt wordt door de dynamische geheugenbeheerder is vaak een bron van een aanval op deze kwetsbaarheden. Alle bestaande tegenmaatregelen met lage impact op performantie maken gebruik van magische waarden, kanaries of andere probabilistische waarden die geheim moeten blijven. In het geval van magische waarden wordt een geheime waarde geplaatst vóór een cruciale geheugenlocatie en door toezicht te houden of de waarde is veranderd, kunnen overlopen opgespoord worden. Als aanvallers willekeurige geheugenlocaties kunnen lezen, dan kunnen ze deze tegenmaatregel omzeilen. Deze tegenmaatregel presenteert een aanpak die, wanneer toegepast op een memory allocator, zal beschermen tegen deze aanvalsvector zonder toevlucht te nemen tot magie. We hebben deze aanpak geïmplementeerd door het wijzigen van een bestaande algemeen gebruikte geheugebeheerder. Uit testen blijkt dat deze tegenmaatregel een te verwaarlozen, soms zelfs positieve, invloed op performantie heeft.

# 1 Inleiding

Sinds de komst van multi-gebruiker systemen, is veiligheid een belangrijk aandachtspunt geworden. In de vroege dagen van de multi-gebruiker systemen, poogden gebruikers om extra tijd op, of de toegang tot, gedeelde bronnen te verkrijgen. Dit was aanleiding tot een heel domein van onderzoek naar computerbeveiliging dat een integraal onderdeel is geworden van het gebied van de computerwetenschappen. De komst van massaal genetwerkte systemen zoals het Internet heeft de behoefte aan veiligheid een hernieuwde urgentie gegeven. Onderzoekers zijn erg succesvol in het ontwerpen van veiligheidsmechanismen die essentiële bescherming bieden aan computersystemen. Het domein van de toegangscontrole, bijvoorbeeld, heeft uitgebreide modellen voor het verlenen van toegangscontrole uitgewerkt, afhankelijk van de specifieke toegangscontrole behoeften van een entiteit. Onderzoek naar cryptografie heeft ook uitgebreide en bewijsbaar veilige cryptografische algoritmes ontwikkeld die ervoor zorgen dat belangrijke gegevens onleesbaar zolang de sleutel veilig is.

Hoewel zulke belangrijke beveiligingsproblemen zijn aangepakt en nog steeds worden verbeterd, blijft er een zeer belangrijk probleem met de implemenatie van de programma's. Vaak zullen deze implementatiefouten de veiligheid die door toegangscontrole en cryptografische protocols aangeboden worden ongedaan maken. In veel gevallen, zullen fouten bij de uitvoering van een cryptografisch protocol het algoritme aanzienlijk verzwakken of zullen fouten in de implementatie van toegangscontrolemechanismen gebruikers de mogelijkheid geven om hun privileges op te waarderen.

Een belangrijke vorm van implementatiefouten die leiden tot beveiligingsproblemen zijn ook het gevolg van een gebrek aan invoercontrole. Dit gebrek aan invoercontrole kan resulteren in een probleem waarbij gegevens worden geïnterpreteerd als computer code. Dit kan zowel voor geïnterpreteerde talen als voor gecompileerde talen gebeuren. Deze kwetsbaarheden zijn echter zeer erg voor programma's die zijn geschreven in C-achtige talen, omdat deze programma's vaak gebruikt worden voor netwerk diensten, besturingssystemen en stuurprogramma's. In de C-achtige talen, kan onvoldoende invoerscontrole leiden tot een bufferoverloop, waar een kwetsbaar programma voorbij het einde van een object, schrijft, waardoor het aangrenzende objecten zal overschrijven.

Deze situatie is verbeterd met de komst van veilige talen, zoals Java en C#, waarmee programmeurs minder directe toegang tot het geheugen hebben en dus voorkomen kan worden dat bepaalde bugs voorkomen. Geheugensanerende talen, laten programmeurs niet toe om handmatig geheugen vrij te geven, wat het probleem van zwevende wijzers verwijdert. Veilige talen zullen meestal niet toestaan dat de programmeur wijzers direct manipuleert en zullen ook niet toestaan dat er rekenkundige operaties op wijzers gebeuren. wat een belangrijke oorzaak van bufferoverlopen elimineert. Toegang tot reeksen via een index worden vaak tijdens looptijd gecontroleerd door de grootte van de reeks bij te houden en ervoor te zorgen dat de toegang binnen de grenzen van deze reeks blijft.

Het is echter niet altijd mogelijk om gebruik te maken van deze veilige talen. Er is een aanzienlijke hoeveelheid van bestaande code die geschreven is in C-achtige talen die hedentendage nog in gebruik is. Bovendien zijn er veel programmeurs die expertise hebben in deze talen en ze nog steeds gebruiken voor het ontwikkelen van nieuwe producten. In sommige gevallen is het gebruik van een C-achtige taal een noodzaak: voor specifieke systeemsoftware, hebben programmeurs directe toegang tot het geheugen nodig, wat door veilige talen belemmerd zou worden. Sommige apparaten hebben ook zeer specifieke beperkingen met betrekking tot het geheugenverbruik en performantie, waardoor het gebruik van een C-achtige taal een aantrekkelijke keuze wordt.

Kwetsbaarheden die het gevolg zijn van fouten in het geheugenbeheer in C-achtige talen, zijn een belangrijke bedreiging voor de veiligheid van huidige computersystemen. De meeste van deze kwetsbaarheden vloeien voort uit het verkeerd gebruik van reeksen, wat resulteert in bufferoverloopkwetsbaarheden. Het misbruik van een dergelijke kwetsbaarheid, kan een aanvaller toelaten om geheugenlocaties, waar de uivoeringsomgeving op berust voor de correcte uitvoering van programma's, te overschrijven.

Dit probleem doet zich voor omdat abstracties die bestaan in een hogere taal niet bestaan in een lager niveau representatie van het programma [Erl07]. Wanneer een C-programma gecompileerd wordt, zal de compiler een aantal mechanismen invoeren die het uitvoeren van programma's vergemakkelijken. Het zal hiervoor deze hoger-niveau abstracties implementeren. Deze mechanismen zijn niet aanwezig in de C-taal en de programmeur heeft hier geen rechtstreeks toegang toe. Omdat C een onveilige taal is, is het echter mogelijk voor een programma om toegang te krijgen tot één van deze mechanismen. Hetzij rechtstreeks door manipulatie van wijzers of per ongeluk door een kwetsbaarheid. Wanneer een dergelijke kwetsbaarheid optreedt, kan een aanvaller hier gebruik van maken om controle te krijgen over de uitvoeringsstroom van een programma. Een voorbeeld van een dergelijk mechanisme is het terugkeeradres: dit adres wordt gebruikt om functies te kunnen uitvoeren. Wanneer een functie wordt uitgevoerd, wordt het adres van de volgende instructie na de functie-oproep geplaatst op de stapel. Zodra de functie klaar is met uitvoeren, zal de uitvoering van het programma worden hervat op het terugkeeradres. Dit mechanisme maakt het mogelijk om programma's uit te voeren met geneste en recursieve functie-oproepen. Een aanvaller kan dit mechanisme misbruiken door gebruik te maken van een kwetsbaarheid die toelaat om het het terugkeeradres naar een andere locatie te doen wijzen. Wanneer de functie eindigt, zal de controle naar deze nieuwe locatie worden overgedragen en zullen de gegevens die op deze locatie zijn opgeslagen als machinecode worden uitgevoerd. Als aanvallers dus een invoer aan het programma kunnen geven dat zal worden opgeslagen in het geheugen van het programma. Als ze bovendien in staat zijn om door middel van een kwetsbaarheid het terugkeeradres te wijzigen, dan zijn ze in staat om willekeurige code uit te voeren met het privilegeniveau van het proces .

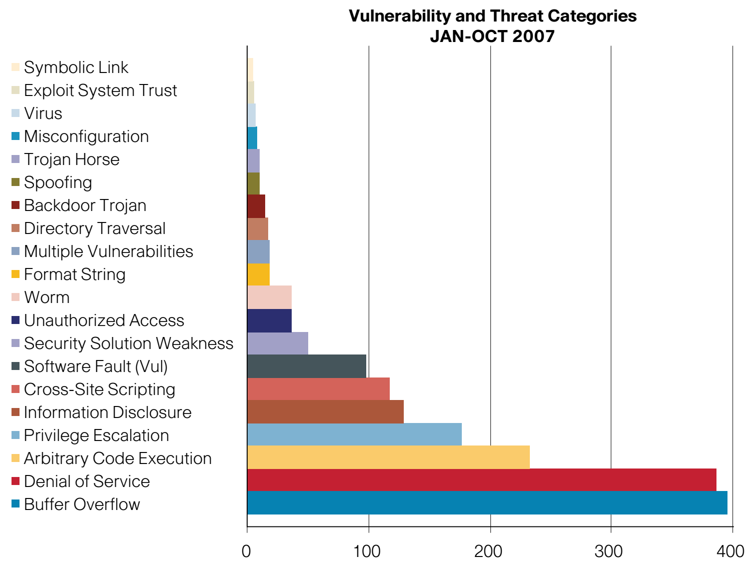
Dit soort kwetsbaarheid werd uitgebuit door veel van de meest verwoestende wormen in de recente geschiedenis: de worm Code Red, die heeft geleid tot een wereldwijde economische verlies geraamd op $2,62 miljard [Eco02]; de Sasserworm, die er voor heeft gezorgd dat Delta Airlines verschillende transatlantische vluchten moest annuleren en verschillende Röntgenmachines van een Zweeds ziekenhuis tijdelijk onbruikbaar maakte [Wikb]; en de Zotob-worm, die waarschijnlijk een nationale crash van computers van het US-VISIT programma van Amerikaanse Departement van Vaderlandsveiligheid heeft veroorzaakt op 18 augustus 2005, wat zorgde voor lange wachtrijen van reizigers op verschillende grote luchthavens [vUCP06].

Volgens het Nationale Kwestbaarheids Database [Nat] van het Amerikaanse Nationale Instituut voor Wetenschap en Technologie, werden er 584 bufferoverloopkwetsbaarheden gemeld in 2005, wat 12 % uitmaakt van de 4852 kwetsbaarheden die gemeld werden dat jaar. In 2004 werden er 341 (14 % van 2352) bufferoverloopkwestbaarheden gemeld. Dit betekent dat, alhoewel het totaal aantal van de gerapporteerde kwetsbaarheden bijna verdubbeld is in 2005, bufferoverloopkwetsbaarheden nog steeds een belangrijke aanvalsbron zijn. 418 van de 584 bufferoverlopen in 2005 had ook een hoog niveau van ernst, wat 21 % van de 1923 kwetsbaarheden met een hoog niveau van ernst is. Zij maken ook 42 % uit van de kwetsbaarheden die een aanvaller beheerdertoegang kunnen geven tot een systeem.

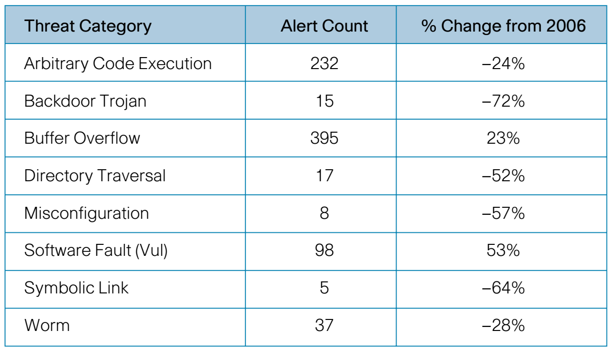
In jaarverslag voor 2007 van Cisco [Inc] worden statistieken over kwetsbaarheden en de oorzaken ervan gepresenteerd op basis van de meldingen van de Cisco Security IntelliShield Alert Manager Dienst [1]. In de grafiek in Figuur 1 worden de kwetsbaarheden en hun effecten vermengd, waardoor het moeilijker is om er nuttige gegevens uit te halen. Het toont echter wel duidelijk aan dat kwetsbaarheden die het mogelijk maken om willekeurige code uit te voeren als ze uitgebuit worden, de tweede grootste fractie van kwetsbaarheden is die gerapporteerd zijn door de IntelliShield Alert Dienst in 2007. Het geeft ook een duidelijke indicatie dat bufferoverloopkwetsbaarheden nog steeds een zeer belangrijke kwetsbaarheid zijn, deze zijn de grootste groep van kwetsbaarheden die gerapporteerd zijn door de IntelliShield Alert Service.

In het verslag wordt ook een vergelijking gemaakt van de trends met kwetsbaarheden in 2006 (zie figuur 2). Hoewel er minder kwetsbaarheden werden gemeld die hebben geresulteerd in de uitvoering van code, werden meer bufferoverlopen gemeld. Dit zou kunnen betekenen dat er meer tegenmaatregelen worden toegepast, die bufferoverlopen in dienstweigeringsaanvallen omzet, in plaats van in code injectie aanvallen. Hoewel dit een positieve trend toont die de positieve resultaten van het onderzoek in dit gebied aantoont, is het ook verontrustend dat de relatieve omvang van bufferoverlopen is toegenomen en een dergelijk groot deel uitmaken

[1]Dit is een dienst die informatie verzamelt over huidige kwetsbaarheden die het distribueert naar klanten die zich abonneren op de service.



Figuur 1: Top 20 Threats and Vulnerabilities, January through October 2007 (source: [Inc], ©Cisco Systems Inc.)



| Threat Category | Alert Count | % Change from 2006 |
|---|---|---|
| Arbitrary Code Execution | 232 | –24% |
| Backdoor Trojan | 15 | –72% |
| Buffer Overflow | 395 | 23% |
| Directory Traversal | 17 | –52% |
| Misconfiguration | 8 | –57% |
| Software Fault (Vul) | 98 | 53% |
| Symbolic Link | 5 | –64% |
| Worm | 37 | –28% |

Figuur 2: Shifts in Threats and Vulnerabilities Reported (source: [Inc], ©Cisco Systems Inc.)

van de gerapporteerde kwetsbaarheden.

## 2 Kwetsbaarheden

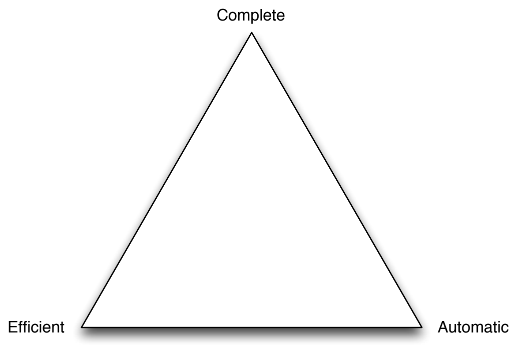
In dit proefschrift concentreren wij ons op enkele belangrijke kwetsbaarheden die gebruikt kunnen worden door een aanvaller om code-uitvoering te bekomen in programma's geschreven in C-achtige talen:

**Buffer overlopen** treden op wanneer een programma niet verzekert dat een schrijfoperatie niet zal schrijven voorbij het einde van het geheugen dat is toegewezen aan een object. Wanneer dit gebeurt, zal aanliggende informatie overschreven worden. Deze bufferoverloopkwetsbaarheden zijn vooral aanwezig in string operaties op reeksen van tekens die zijn opgeslagen op de stapel. Meestal zal een aanvaller deze exploiteren door een terugkeeradres op de stapel te overschrijven om zo code uitvoering te bekomen. Dit soort van kwetsbaarheid kan echter ook bestaan in andere regio's van het geheugen: op de hoop en in de data / bbs[2] sectie.

**Zwevende wijzers** treden op wanneer wijzers bestaan naar geheugen dat is vrijgegeven. Zo een wijzer kan verwijzen naar een geheugenlocatie die vrijgegeven is, hetzij expliciet door de programmeur (bijvoorbeeld door de functie vrijgeven op te roepen) of door code die wordt gegenereerd door de compiler (bijvoorbeeld een epiloogfunctie, waar het stapelkader van de functie wordt verwijderd van de stapel). Het bestaan van dergelijke zwevende wijzers is echter geen veiligheidsprobleem zolang deze wijzers niet worden gebruikt. Als een programma gebruik maakt van één van deze wijzers, wordt geheugen aangesproken dat niet meer bevat wat de programmeur verwacht. Dit kan een veiligheidsprobleem veroorzaken omdat het geheugen kan zijn hergebruikt. Als een programma via een zwevende wijzer schrijft naar geheugen dat hergebruikt is zal het de informatie die er is opgeslagen overschrijven, denkende dat het een bepaald object is. Een ander deel van het programma zou dit geheugen tegelijkertijd ook als een ander object kunnen gebruiken, waardoor prolbemen kunnen veroorzaakt worden.

**Formaat string kwetsbaarheden** zijn kwetsbaarheden die optreden wanneer formaatfuncties verkeerd gebruikt worden door een programmeur. Formaatfuncties zijn functies die een variabel aantal argumenten samen met een formaat string als argument verwachten.Deze formaat string zal aangeven hoe de formaatfunctie zijn weergave zal formateren. De formaat string is een tekenreeks die letterlijk gekopieerd wordt naar de weergave stroom tenzij een % karakter wordt tegengekomen. Dit karakter wordt gevolgd door een formaat

[2]blok begonnen door symbool



Figuur 3: Countermeasure triangle

specificeerder die de weergave zal manipuleren. Wanneer een formaat specificeerder een argument vereist, verwacht de formaat functie dit argumentt e vinden op de stapel (bijvoorbeeld de volgende oproep: $printf("\%d" d)$), hier verwacht printf de integer d als tweede argument voor de printf oproep op de te stapel te vinden). Een formaat string kwetsbaarheid treed op als een aanvaller controle heeft over de formaat string (bijvoorbeeld $printf(s)$, waarbij $s$ door de gebruiker geleverd werd). De aanvaller is nu in staat te controleren wat de functie van de stapel en kan er voor zorgen dat het programma naar willekeurige locaties in het geheugen schrijft.

**Getalfouten** [ble02, You03] zijn niet uitbuitbaar op zichzelf, maar de uitbuiting van dit soort fouten kan leiden tot een situatie waarin het programma kwetsbaar wordt voor één van de eerder beschreven kwetsbaarheden. Twee soorten integer fouten die kunnen leiden tot de exploiteerbaarheid van kwetsbaarheden bestaan: overlopen van een getal en tekenfouten.

Veel tegenmaatregelen zijn ontworpen om te beschermen tegen uitbuiting van deze kwetsbaarheden. Deze tegenmaatregelen worden meestal beoordeeld aan de hand van drie belangrijke criteria: volledigheid, efficiëntie en het niveau van automatisering. In analogie met het project driehoek in het technische ontwerp " Goed, Goedkoop, Betrouwbare: Kies er twee "[Wika], de meeste tegenmaatregelen zullen alleen voldoen aan twee van de drie criteria.

**Volledigheid** bepaalt hoe volledig de bescherming is tegen de kwetsbaarheid die wordt geaddresseerd.

**Efficiëntie** bepaalt het effect dat de tegenmaatregel heeft op de prestatie en het geheugengebruik.

**Automatisch** omvat meerdere facetten van de tegenmaatregelen die menselijke tussenkomst zouden kunnen vereisen. Hoe minder handmatige tussenkomst vereist is, hoe meer automatische de tegenmaatregel zal worden. Als de tegenmaatregel niet compatibel is met de bestaande C-code, dan zal handmatige interventie noodzakelijk zijn. Tegenmaatregelen die de interpretatie en de verwerking van de resultaten vereisen (zoals statische en dynamische analyzers), zullen ook enige manuele interventie vereisen.

Elke tegenmaatregel is een compromis tussen efficiëntie, volledigheid en automatisering. Vaak zal aan twee van deze eigenschappen worden voldaan, maar moeten compromissen worden gemaakt in het andere veld:

· Een tegenmaatregel die efficiënt en volledig is zal handmatige interventie nodig hebben om toegepast te worden op een programma.

· Een tegenmaatregel die volledig is en automatisch kan worden toegepast, zal inefficiënt zijn.

· Een tegenmaatregel die efficiënt is en automatisch kan worden toegepast, zal onvolledig zijn.

Dit is echter de ideale situatie bij tegenmaatregelen: soms is het mogelijk om verbetering te bekomen in één gebied, terwijl er slechts minimale of geen verlies wordt gemaakt in een ander. De tegenmaatregelen in dit proefschrift zijn van die aard: ze zijn een aanzienlijke verbetering van de volledigheid van een tegenmaatregel, terwijl er slechts weinig (of helemaal geen) verlies is op het gebied van efficiëntie.

## 3 Tegenmaatregel voor stapel-gebaseerde bufferoverlopen

Een belangrijke bijdrage van dit proefschrift is een tegenmaatregel voor stapel-gebaseerde bufferoverlopen. In deze tegenmaatregel worden controle stroom gegevens (gegevens die gebruikt worden voor het regelen van de controle stroom van het programma, zoals een terugkeeradres) gescheiden van reguliere gegevens op de stapel. Deze scheiding wordt bereikt door het toewijzen van twee waarden voor elk type van de gegevens: doelwaarde (hoe waardevol is dit soort gegevens voor een aanvaller tijdens een poging om een code injection aanval uit te voeren) en bronwaarde (hoe groot is de kans is dat een aanvaller dit type van gegevens kan misbruiken voor het uitvoeren van een aanval). Bijvoorbeeld, het terugkeeradres (en andere opgeslagen registers) heeft een hoge doelwaarde, aangezien een aanvaller die er controle over heeft deze kan gebruiken voor het uitvoeren van code injectie. Het terugkeeradres heeft ook een lage bronwaarde; aanvallers zullen er nooit directe controle over hebben, ze hebben een aanval nodig om het te wijzigen. Een reeks van karakters heeft een lage doelwaarde gezien aanvallers in het algemeen geen code injectie verkrijgen bij het overschrijven van een dergelijke reeks. Een reeks van karakters heeft wel een hoge bronwaarde: deze reeksen zijn in het algemeen degene die kwetsbaar zijn voor bufferoverlopen. Afhankelijk van het soort gegevens kan de stapel dan worden opgesplitst in meerdere stapels die van elkaar gescheiden zijn. Als gevolg kan een bufferoverloop in een reeks van karakters alleen andere reeksen van characters overschrijven, maar geen terugkeeradressen. Deze tegenmaatregelen is zeer efficiënt, automatisch en completer dan de tegenmaatregelen die zich richten op deze eerste twee eigenschappen.

## 4 Tegenmaatregel voor hoop-gebaseerde bufferoverlopen

Een ander belangrijk tegenmaatregel die is ontworpen en geïmplementeerd maakt het moeilijker voor een aanvaller om een code injection aanval uit te overen als er een hoop-gebaseerde bufferoverloop bestaat. Wanneer een hoop-gebaseerde bufferoverloop wordt uitgebuit, zal een aanvaller de geheugenbeheersinformatie wijzigen om op een betrouwbare manier code injectie te verkrijgen. De tegenmaatregel voorkomt dit soort aanvallen door het scheiden van geheugenbeheersinformatie van de rest van het dynamisch toegewezen geheugen. De tegenmaatregel is ook zeer efficiënt, automatisch en completer dan andere tegenmaatregelen die zich richten op deze eerste twee eigenschappen.

## 5 Conclusie

### 5.1 Toekomstig werk

De auteur van dit proefschrift is momenteel betrokken bij het ontwerpen en implementeren van een aantal nieuwe tegenmaatregelen die betere bescherming kunnen bieden tegen de kwetsbaarheden die besproken werden in dit proefschrift. We bespreken hier twee tegenmaatregelen, terwijl een derde besproken wordt in de volgende sectie.

#### 5.1.1 Een grenscontroleur voor rekenkundige operaties op wijzers

Tijdens een onderzoeksverblijf aan de Universiteit van Stony Brook, heeft de auteur een derde tegenmaatregel ontworpen dat grenscontroles doet voor C. Dit wordt bereikt door de invoering van extra controles bij rekenkundige operaties op wijzers. Huidige grenscontroleurs voeren meestal hun controles uit wanneer een wijzer gebruikt wordt. Door deze controle op berekeningstijd in plaats van bij gebruik te doen zou een verhoging van performantie bereikt moeten worden. Wanneer een object word gealloceerd zal het gehele geheugen van dat object gemarkeerd worden met een unieke waarde, een etiket genaamd. Wanneer een rekenkundig operatie wordt uitgevoerd op een wijzer die wijst naar een object, wordt het etiket van het object waar de wijzer naar wijst vergeleken met het etiket van het resultaat van de rekenkundige operatie. Als ze niet gelijk zijn, dan is een wijzer gemaakt, dat buiten de grenzen van het object wijst. Dit werk is nog aan de gang, maar we verwachten dat de tegenmaatregel volledig zal zijn en verenigbaar met de bestaande code. Onze focus is vooral op het maken van grenzencontroleurs die efficiënter en schaalbaar zijn, zodat ze kunnen worden gebruikt in productiesystemen.

### 5.1.2 Een tegenmaatregel voor zwevende wijzers

Deze tegenmaatregel is gebaseerd op de ervaring die werd opgedaan bij het ontwikkelen van de grenzencontrolleur, maar concentreert zich op zwevende wijzers. Bestaande tegenmaatregelen voor zwevende wijzers [DA06] lijden aan een hoge performantie-impact omdat veel controles moeten worden uitgevoerd om te voorkomen dat zwevende wijzers gebruikt worden. De tegenmaatregel die zal worden ontwikkeld, zal naar verwachting efficiënter zijn door een associatie van het object naar de wijzer bij te houden. Wanneer een wijzer wordt ingesteld om te verwijzen naar een bepaald object, zal ook een associatie van het object naar de wijzer wordt bijgehouden. Wanneer een object vrijgegeven wordt, zal de associatie gebruikt worden om alle wijzers die verwijzen naar dit object ongeldig te maken. Wanneer het programma een dergelijke wijzer probeert te gebruiken zal dit een crash tot gevolg hebben, wat voorkomt dat de wijzer op een ongeldige wijze gebruikt wordt.

## 5.2 Toekomstig onderzoeksmogelijkheden en toepassingsdomeinen

In deze rubriek bespreken we onderzoeksmogelijkheden en uitdagingen van een aantal nieuwe en recente technologieën met betrekking tot het ontwerp van nieuwe tegenmaatregelen. We bespreken ook nieuwe toepassingsgebieden voor tegenmaatregelen voor code injectie aanvallen.

### 5.2.1 Ingebedde systemen en mobiele apparaten

Nieuwe technologieën zoals ingebedde systemen, sensornetwerken en mobiele apparaten zorgen voor nieuwe uitdagingen. Deze systemen moeten vaak werken met zeer beperkte middelen, waardoor het gebruik van C vaak een noodzaak is, wat resulteert in dezelfde problemen als in de traditionele architecturen. Vanwege de specifieke eisen van deze systemen, zullen de tegenmaatregelen ook verschillen van de tegenmaatregelen voor de traditionele systemen, hoewel sommige van dezelfde basisideeën overgezet kunnen worden naar deze apparaten. Er bestaan vele soorten van ingebedde aparaten, die op een groot aantal verschillende architecturen draaien. Terwijl de grote hoeveelheid architecturen het moeilijker maakt voor aanvallers om kwetsbaarheden uit te buiten, zorgt dit ook voor extra complexiteit bij het ontwikkelen van tegenmaatregelen.

In augustus 2006 werden een aantal kwetsbaarheden [Orm06a, Orm06b] ontdekt in LibTIFF[3]. LibTIFF wordt gebruikt in een aantal desktop-besturingssystemen, zoals Linux en Mac OS X. Het wordt ook gebruikt op de Apple iPhone, waar deze kwetsbaarheid op grote schaal werd misbruikt door de gebruikers van de iPhone [Moo] voor het uitvoeren van een "gevangenisuitbraak "[4]. Deze kwetsbaarheid kan uitgebuit worden via zowel MobileMail (het mailprogramma van de iPhone) en MobileSafari (de iPhone webbrowser) en als gevolg kan deze kwetsbaarheid van op afstand uitgebuit worden door een gebruiker naar een webstek te laten surfen waarop een specifiek TIFF-bestand te vinden is of door een dergelijk bestand naar de gebruiker te e-mailen.

Deze kwetsbaarheid was ook aanwezig op een ander mobiel toestel: de Sony PlayStation Portable, waar de kwetsbaarheid ook gebruikt werd om functionaliteit aan te bieden buiten de goedkeuring van de fabrikant om. In dit geval werd de kwetsbaarheid gebruikt om meer bevoegdheden te bekomen zodat gebruikers zelfgemaakte spelletjes konden spelen.

Deze voorbeelden tonen aan dat software die oorspronkelijk ontworpen is voor bureau-omgevingen op grote schaal overgezet wordt naar deze nieuwe apparaten, wat resulteert in de aanwezigheid van gelijkaardige kwetsbaarheden in deze apparaten. Naarmate meer en meer van deze apparaten op de markt komen, zullen vergelijkbare kwetsbaarheden ontdekt en uitgebuit worden.

Momenteel bestaan er zeer weinig tegenmaatregelen voor ingebedde systemen. De belangrijkste tegenmaatregel die momenteel is ingezet op deze systemen is de stapelkoekbescherming in Windows CE 6. Deze tegenmaatregel is gebaseerd op de StackGuard tegenmaatregel [CPM+98], waar een geheim random getal voor het terugkeeradres van de functie wordt geplaatst bij het starten van de functie en waarvoor de terugkeer wordt bevestigd dat het getal niet veranderd is. De achterliggende gedachte is dat een aanvaller het terugkeeradres niet zallen kunnen wijzigen zonder ook het random getal te wijzigen. Dit getal moet wel geheim blijven, anders kan een aanvaller het getal gewoon herschrijven bij het uitbuiten van een bufferoverloop. Dit is een inherente zwakte in deze benadering.

Veel van de tegenmaatregelen die momenteel in gebruik zijn op de desktop systemen kunnen overgezet worden naar ingebedde apparaten. Vanwege het be-

[3]LibTIFF is een bibliotheek voor het lezen en schrijven van TIFF-bestanden, een populair formaat voor het opslaan van afbeeldingen

[4]Standaard is het niet mogelijk voor gebruikers om extra applicaties op de iPhone te installeren. De term gevangenisuitbraak verwijst naar het omzeilen van deze beperkingen, waardoor gebruikers in staat zijn om de volledige controle over het apparaat te bekomen.

perkte geheugen en de verwerkingskracht is efficiëntie echter een belangrijke bezorgdheid bij deze apparaten. Verschillende andere beperkingen in de architectuur kunnen ook een probleem zijn bij het overzetten van tegenmaatregelen naar deze architecturen. Bijvoorbeeld, veel architecturen hebben geen ondersteuning voor paginering, waardoor het moeilijk wordt om een tegenmaatregel zoals Adresruimte Lay-out Randomisatie (ALR) op deze apparaten over te zetten.

Een ander belangrijke beperking bij het overzetten van tegenmaatregelen naar ingebedde apparaten is het feit dat vele tegenmaatregelen werden ontworpen met een specifieke architectuur of besturingssysteem in gedachten. Dit kan het overzetten moeilijker en gevoeliger voor omzeiling maken. Als een tegenmaatregel wordt overgezet, dan zal de tegenmaatregelontwikkelaar moet zorgen dat de tegenmaatregel niet gemakkelijk omzeild kan worden op het nieuwe platform. Een voorbeeld waarbij een dergelijke overzetting fout is gegaan is wanneer Microsoft de StackGuard tegenmaatregel [CPM$^+$98] van GCC naar Visual Studio [Bra03] heeft overgezet. De details van het Windows-besturingssysteem werden niet in acht genomen, hetgeen resulteerde in het feit dat aanvallers de tegenmaatregel konden omzeilen door het terugkeeradres te negeren en verder te schrijven op de stack, totdat ze de functiewijzers die gebruikt worden voor de behandeling van uitzonderingen konden overschrijven. Vervolgens werd een uitzondering gegenereerd door de aanvallers en kon hun geïnjecteerde code uitgevoerd [Lit03]. Een mogelijke manier om dit soort problemen te voorkomen bij het overzetten van tegenmaatregelen naar een nieuw platform is door gebruik te maken van een machinemodel voor tegenmaatregelen [YJP05].

### 5.2.2 Virtuele machine monitors

Andere technologieën als virtuele machine monitors kunnen ook de basis vormen voor het ontwerpen van nieuwe tegenmaatregelen. Vanwege de opkomst van de efficiënte virtuele machines, is het mogelijk om tegenmaatregelen op een lager niveau uit te werken dan voorheen mogelijk was: als er een programma wordt uitgevoerd in een virtuele machine is het mogelijk om de hele architectuur te wijzigen om een beveiligingsprobleem op te lossen, ipv. rond specifieke problemen van een bepaalde architectuur te moeten werken. Dit kan nuttig zijn in het ontwerpen van prototypes die kunnen worden toegepast op toekomstige herzieningen van de architectuur of toegepast kunnen worden in een omgeving waarin virtuele machine monitors gebruikt worden.

We zijn momenteel in het proces van het ontwerpen en implementeren van een tegenmaatregel die gebruik maakt van een virtuele machine monitor om te beschermen tegen de kwetsbaarheden die besproken werden in dit proefschrift. De tegenmaatregel is gebaseerd op de observatie dat de meeste uitbuitingen die leiden tot uitvoering van code, een beroep doen op het wijzigen van de waarde van een wijzer. In deze tegenmaatregel worden alle wijzers opgeslagen in geheugen dat enkel leesbaar is wat voorkomt dat deze rechtstreeks door een kwetsbaarheid worden gewijzigd. Wanneer het programma schrijft naar een wijzer, genereert de compiler een speciale instructie die kan worden gebruikt voor het schrijven naar het geheugen van deze locatie. Deze instructie wordt dan geëmuleerd door de virtuele machine monitor. Als een kwetsbaarheid bestaat in een programma waarmee een aanvaller willekeurige geheugenlocaties kan overschrijven dan kan deze geen wijzers wijzigen, omdat alleen de speciale instructie kan schrijven naar wijzers. Aanvallers zijn niet in staat zijn om deze speciale instructie te genereren totdat zij in staat zijn om code te uitvoeren. Deze tegenmaatregel zou een significante verbetering van de bescherming tegen uitbuiting van deze kwetsbaarheden zijn. Zodra het prototype heeft nagegaan of deze aanpak haalbaar is of niet, zou het gebruikt kunnen worden als basis voor mogelijke latere herzieningen van de architectuur.

## Referenties

[ble02] blexim. Basic integer overflows. *Phrack*, 60, December 2002.

[Bra03] Brandon Bray. Security improvements to the whidbey compiler. `http://weblogs.asp.net/branbray/archive/2003/11/11/51012.aspx`, November 2003.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[DA06] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, Philadelphia, Pennsylvania, U.S.A., 2006. IEEE Computer Society.

[Eco02] Computer Economics. 2001 economic impact of malicious code attacks. `http://www.computereconomics.com/article.cfm?id=133`, January 2002.

[Erl07] Úlfar Erlingsson. Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research, November 2007.

[Inc] Cisco Systems Inc. Cisco 2007 annual security report. `http://www.cisco.com/web/about/security/cspo/docsCisco2007Annual_Security_Report.pdf`.

[Lit03] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf, September 2003.

[Moo] HD Moore. Cracking the iphone. http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html.

[Nat] National Institute of Standards and Technology. National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

[Orm06a] Tavis Ormandy. Libtiff next rle decoder remote heap buffer overflow vulnerability. http://www.securityfocus.com/bid/19282, Aug 2006.

[Orm06b] Tavis Ormandy. Libtiff tifffetchshortpair remote buffer overflow vulnerability. http://www.securityfocus.com/bid/19283, Aug 2006.

[vUCP06] Kevin Poulson vs. U.S Customs and Border Protection. Declaration of shari suzuki in opposition to motion for summary judgement. http://wiredblogs.tripod.com/27BStroke6/suzukidecl.pdf, May 2006.

[Wika] Wikipedia. Project triangle. http://en.wikipedia.org/wiki/Project_triangle.

[Wikb] Wikipedia. Wikipedia entry for the sasser worm. http://en.wikipedia.org/wiki/Sasser.

[YJP05] Yves Younan, Wouter Joosen, and Frank Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005)*, College Park, Maryland, U.S.A., March 2005. IEEE, IEEE Press.

[You03] Yves Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003.