Licentiate thesis

Policy and Implementation Assurance for Software Security

by
John Wilander

LiU-Tek-Lic-2005:62

2005-10-17

Licentiate thesis

Policy and Implementation Assurance for Software Security

by **John Wilander**

LiU-Tek-Lic-2005:62

Advisor : **Professor Mariam Kamkar**
Dept. of Computer and Information Science
at Linköpings universitet

Opponent : **Doctor Andrei Sabelfeld**
Department of Computer Science
at Chalmers University of Technology

Avdelning, Institution
Division, Department

LINKÖPINGS UNIVERSITET

SaS,
Dept. of Computer and Information Science
581 83 LINKÖPING

Datum
Date

2005-10-17

Språk
Language

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ ________

Rapporttyp
Report category

- ☒ Licentiatavhandling
- ☐ Examensarbete
- ☐ C-uppsats
- ☐ D-uppsats
- ☐ Övrig rapport
- ☐ ________

ISBN
91-85457-65-5

ISRN
LiU-Tek-Lic-2005:62

Serietitel och serienummer
Title of series, numbering

ISSN
0280-7971

URL for electronic version
`http://www.ida.liu.se/~johwi/research_publications/licentiate_thesis.pdf`

Titel [No Swedish title]

Title Policy and Implementation Assurance for Software Security

Författare
Author
John Wilander

Sammanfattning
Abstract

To build more secure software, accurate and consistent security requirements must be specified. We have investigated current practice by doing a field study of eleven requirement specifications on IT systems. The overall conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions.

To build more secure software we specifically need assurance requirements on code. A way to achieve implementation assurance is to use effective methods and tools that solve or warn for known vulnerability types in code. We have investigated the effectiveness of four publicly available tools for run-time prevention of buffer overflow attacks. Our comparison shows that the best tool is effective against only 50 % of the attacks and there are six attack forms which none of the tools can handle. We have also investigated the effectiveness of five publicly available compile-time intrusion prevention tools. The test results show high rates of false positives for the tools building on lexical analysis and low rates of true positives for the tools building on syntactical and semantical analysis.

As a first step toward a more effective and generic solution we propose dependence graphs decorated with type and range information as a way of modeling and pattern matching security properties of code. These models can be used to characterize both good and bad programming practice. They can also be used to visually explain code properties to the programmer.

Nyckelord
Keywords
Software security, security requirements, intrusion prevention

Abstract

To build more secure software, accurate and consistent security requirements must be specified. We have investigated current practice by doing a field study of eleven requirement specifications on IT systems. The overall conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions.

To build more secure software we specifically need assurance requirements on code. A way to achieve implementation assurance is to use effective methods and tools that solve or warn for known vulnerability types in code. We have investigated the effectiveness of four publicly available tools for run-time prevention of buffer overflow attacks. Our comparison shows that the best tool is effective against only 50 % of the attacks and there are six attack forms which none of the tools can handle. We have also investigated the effectiveness of five publicly available compile-time intrusion prevention tools. The test results show high rates of false positives for the tools building on lexical analysis and low rates of true positives for the tools building on syntactical and semantical analysis.

As a first step toward a more effective and generic solution we propose dependence graphs decorated with type and range information as a way of modeling and pattern matching security properties of code. These models can be used to characterize both good and bad programming practice. They can also be used to visually explain code properties to the programmer.

Keywords : Software security, security requirements, intrusion prevention

Acknowledgments

I would like to thank my advisor Professor Mariam Kamkar for her support in my research, my teaching and my overall work at the Department of Computer and Information Science.

Next, I would like to thank David Byers who on numerous occasions has previewed what I have written and given me important feedback on content and language.

Further, I would like to thank the people at the Programming Environments Laboratory, both fellow Ph.D. students, senior researchers, and our ever so kind administrator Bodil Mattsson Kihlström for providing me with a nice and supporting working environment. I would like to send special thanks to Professor Kristian Sandahl, Jens Gustavsson, Kaj Nykvist, and Andreas Borg for help on previewing my papers, and to Jon Edvardsson for help and support in teaching.

I would also like to thank Professor Nahid Shahmehri and Associate Professor Simin Nadjm-Tehrani for tips and feedback on my research.

This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

John Wilander
Linköping, October 17th, 2005

Contents

1 Introduction **1**

1.1 Thesis Overview 3

2 Security Assurance **5**

2.1 Policy Assurance 6

2.2 Implementation Assurance 6

3 Summary of Papers **9**

3.1 Field Study of Security Requirements 9

3.2 Run-Time Buffer Overflow Prevention 10

3.3 Compile-Time Intrusion Prevention 10

3.4 More Generic Compile-Time Intrusion Prevention 10

4 Related Work **13**

4.1 Run-Time Intrusion Prevention 13

4.1.1 Canary-Based Tools 14

4.1.2 Boundary Checking Tools 15

4.1.3 Tools Copying and Checking Target Data 16

4.1.4 Tools using Randomized Instructions 17

4.1.5 Library Wrappers 17

4.1.6 Non-Executable and Randomized Memory 18

4.2 Related Studies on Attacks and Prevention 19

5 Security Requirements **21**
5.1 Abstract 21
5.2 Introduction 22
5.3 Security Requirements 23
5.3.1 From a RE Point of View 23
5.3.2 From a Security Point of View 24
5.4 Security Testing 25
5.5 Field Study of Eleven Requirements Specifications 26
5.5.1 Systems in the Field Study 26
5.5.2 Detailed Categorization of Security Requirements . . 28
5.5.3 Discussion 31
Security Requirements are Poorly Specified 31
Security Requirements are Mostly Functional 34
Security Requirements Absent 35
5.5.4 Possible Shortcomings 36
5.6 Conclusions 37
5.7 Acknowledgments 37

6 Dynamic Buffer Overflow Prevention **39**
6.1 Abstract 39
6.2 Introduction 40
6.2.1 Scope 41
6.2.2 Paper Overview 42
6.3 Attack Methods 42
6.3.1 Changing the Flow of Control 42
6.3.2 Memory Layout in UNIX 43
6.3.3 Attack Targets 44
6.3.4 Buffer Overflow Attacks 45
6.4 Intrusion Prevention 47
6.4.1 Static Intrusion Prevention 47
6.4.2 Dynamic Intrusion Prevention 48
6.4.3 StackGuard 48
The StackGuard Concept 49
Random Canaries Unsupported 50
6.4.4 Stack Shield 50
Global Ret Stack 51

Ret Range Check 51
Protection of Function Pointers 51
6.4.5 ProPolice 52
The ProPolice Concept 52
Building a Safe Stack Frame 52
6.4.6 Libsafe and Libverify 54
Libsafe 54
Libverify 55
6.4.7 Other Dynamic Solutions 56
6.5 Comparison of the Tools 58
6.6 Common Shortcomings 61
6.6.1 Denial of Service Attacks 61
6.6.2 Storage Protection 62
6.6.3 Recompilation of Code 62
6.6.4 Limited Nesting Depth 62
6.7 Related Work 62
6.8 Conclusions 63
6.9 Acknowledgments 64

7 Static Intrusion Prevention **65**
7.1 Abstract 65
7.2 Introduction 66
7.3 Attacks and Vulnerabilities 68
7.3.1 Changing the Flow of Control 68
7.3.2 Buffer Overflow Attacks 69
7.3.3 Buffer Overflow Vulnerabilities 70
7.3.4 Format String Attacks 70
7.3.5 Format String Vulnerabilities 71
7.4 Intrusion Prevention 72
7.4.1 Dynamic Intrusion Prevention 73
7.4.2 Static Intrusion Prevention 73
7.4.3 ITS4 73
7.4.4 Flawfinder and Rats 74
7.4.5 Splint 75
7.4.6 BOON 76
7.4.7 Other Static Solutions 77

Software Fault Injection 77
Constraint-Based Testing 78
7.5 Comparison of Static Intrusion Prevention Tools 78
7.5.1 Observations and Conclusions 79
7.6 Related Work 80
7.7 Conclusions 82

8 Modeling Security Properties 83
8.1 Abstract 83
8.2 Introduction 84
8.2.1 Paper Overview 85
8.3 Survey of Static Analysis Tools 85
8.3.1 Splint 86
8.3.2 BOON 86
8.3.3 Cqual 86
8.3.4 Metal and xgcc 87
8.3.5 MOPS 88
8.3.6 IPSSA 88
8.3.7 Mjolnir 89
8.3.8 Eau Claire 89
8.3.9 Summary 91
8.4 The Need for Visual Models 91
8.5 The Dual Modeling Problem 92
8.5.1 Modeling Good Security Properties 93
8.5.2 Modeling Bad Security Properties 93
8.6 Ranking of Potential Vulnerabilities 94
8.6.1 Using the Dual Model for Ranking 94
8.7 A More Generic Modeling Formalism 95
8.7.1 Program Dependence Graphs 95
8.7.2 System Dependence Graphs 96
8.7.3 Range Constraints in SDGs 97
8.7.4 Type Information in SDGs 97
8.7.5 Static Analysis Using SDGs 97
8.8 Modeling Security Properties 98
8.8.1 Integer Flaws 98
Integer Signedness Errors. 99

Integer Overflow/Underflow. 100
Integer Input Validation. 101
8.8.2 Modeling Integer Flaws 101
8.8.3 The Double `free()` Flaw 102
8.8.4 Modeling External Input 103
8.9 Future Work 103
8.10 Conclusions 106
8.11 Acknowledgments 106

9 Future Work **107**
9.1 Security Requirements 107
9.2 Run-Time Intrusion Prevention 108
9.3 Compile-Time Intrusion Prevention 109

10 Summary and Conclusions **111**

11 Appendices **113**
.1 Empirical Test of Dynamic Buffer Overflow Prevention . . . 114
.2 Theoretical Test of Dynamic Buffer Overflow Prevention . . 117
.3 Static Testbed for Instrusion Prevention Tools 121

Bibliography **125**

Chapter 1

Introduction

> "When looked on as an absolute, creating a secure system is an ultimate, albeit unachievable, goal."
>
> —Elisabeth C. Sullivan

Security is perhaps one of the most challenging problems programmers face, as pointed out by the editor of Dr. Dobb's Journal in early 2004 [6]. As individuals, organizations, and society rely more and more on computers, networks, and applications, questions of their trustworthiness in terms of security and privacy are asked over and over again. How vulnerable are our computer systems to security attacks? How can we make them more secure?

According to statistics from CERT Coordination Center at Carnegie Mellon University, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software (see Figure 1.1) [7]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [8].

There are many possible remedies to insecurity. Protecting the perimeter of your organization with the use of firewalls, virtual private networks, and intrusion detection is one way. Defining a local security policy and communicating this to staff and users is another. A third approach is building

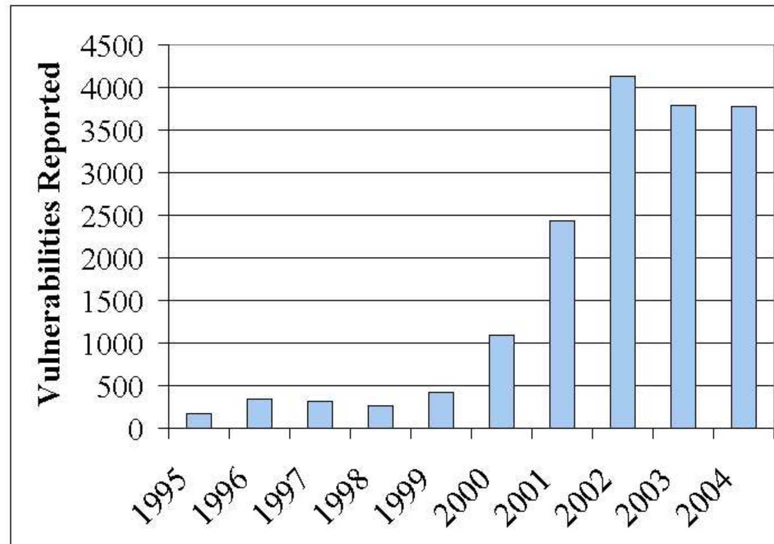


Figure 1.1: Software security vulnerabilities reported to CERT 1995–2004.

or requiring more secure software in the first place. By eliminating security vulnerabilities inside the software, we can take away the attackers' tools and protect the targets they aim for.

For consumers of software the security of the products they use relies heavily on the security requirements specified for the products. If those requirements are poorly specified there is nothing saying that designers, implementers and testers will strive for security. Instead, costs and time will be focused on meeting the other requirements, and security issues may be left for maintenance to take care of in the infamous *penetrate and patch* manner [9].

In the middle of January 2002 the discussion about responsibility for security intrusions took a new turn. The US National Academies released a prepublication recommending that policy-makers create laws that would hold companies accountable for security breaches resulting from vulnerable products [10], which received global media attention [11, 12]. So far, only the intruder can be charged in court. In the future, software companies may be charged for not preventing intrusions. This stresses the importance

of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

In this thesis we address some of the problems and potential solutions in the area of software security assurance. We have studied current practice in security requirements, done comparative studies of security tools and techniques for software development, and finally proposed a new, more generic technique for static security analysis of code.

1.1 Thesis Overview

The rest of this licentiate thesis is organized as follows. Chapter 2 gives definitions of the various categories of software security assurance, with special attention to the two categories covered in this thesis—policy assurance and implementation assurance.

Chapter 3 gives a brief summary of the four papers that make up the major content of the thesis. In Chapter 4 I present related work not covered in the papers. The major part is about run-time intrusion prevention since over two years have past since we published our comparative study of run-time defense.

Chapter 5 to 8 are four full papers published between 2002 and 2005. They cover security requirements, run-time intrusion prevention, compile-time intrusion prevention, and our proposed modeling formalism for more generic compile-time intrusion prevention respectively.

Future work, conclusions and summary can be found in Chapter 9 and 10. Chapter 11 contains appendices for the papers in Chapter 6 to 7. Last is a bibliography of references.

John Wilander
Linköping, October 17th, 2005

Chapter 2

Security Assurance

Requirements on security alone do not solve the problem of insecure systems. There has to be some way of evaluating the product to see if it meets its security requirements. This is what we call *security assurance*, or simply *assurance*. Bishop and Sullivan define security assurance [13]:

Security Assurance — confidence that an entity meets its security requirements, based on specific evidence provided by the application of assurance techniques.

They categorize security assurance into policy assurance, design assurance, implementation assurance, and operational or administrative assurance:

Policy Assurance — evidence establishing that the set of security requirements in the policy is complete, consistent, and technically sound.

Design Assurance — evidence establishing that a design is sufficient to meet the requirements of the security policy.

Implementation Assurance — evidence establishing that the implementation is consistent with the security requirements of the security policy.

Operational and Administrative Assurance — evidence establishing that the system sustains the security policy requirements during installation, configuration, and day-to-day operation.

In this thesis we focus on policy assurance (Chapter 5), and implementation assurance (Chapter 6, 7, and 8).

2.1 Policy Assurance

Policy assurance is crucial for good security in software since it assures that the proper requirements on security are specified. This means both requirements on *security features* such as login, encryption, and logging, and requirements on *secure features* where the right methods and techniques have been used for design, implementation, and operational and administrative assurance.

Requirements on assurance measures can be called *assurance requirements* and should not be mistaken for policy assurance where you evaluate the requirements themselves. Policy assurance checks that you specify the right requirements, assurance requirements are requirements on secure design, implementation, and operation.

2.2 Implementation Assurance

To build more secure software we need requirements on secure code. However, requirements on avoiding known vulnerability types in the code can be hard to evaluate. Another, perhaps more realistic way, to achieve implementation assurance is to use effective methods and tools that solve or warn for known vulnerability types.

Such a starting point for producing more secure software would, or could be tools that can be applied directly to the source code. This means trying to solve the problems in the implementation and testing phase.

Applying security related methodologies throughout the whole development cycle would most likely be more effective, but given the amount of existing software ("legacy code"), the desire for modular design reusing software components programmed earlier, and the time it would take to

educate software engineers in secure analysis and design, we argue that security tools that aim to clean up vulnerable source code (i.e. tools for implementation assurance) are necessary. A further discussion of this issue can be found in the January/February 2002 issue of IEEE Software [14].

Chapter 3

Summary of Papers

In this thesis we focus on policy assurance and implementation assurance. The contents of the Chapters 5, 6, 7, and 8 are four full papers published between 2002 and 2005 [1, 2, 3, 4]. In the following sections we briefly summarize those papers.

3.1 Field Study of Security Requirements

To build more secure software, accurate and consistent security requirements must be specified, forming a security policy for the system. In our paper "Security Requirements—A Field Study of Current Practice" [1] (Chapter 5) we investigate current practice by doing a field study of eleven requirement specifications on IT systems being built 2003 through 2005. To evaluate the outcome we have looked into documentation of security requirements from the requirements engineering community as well as from the security community. Requirements found in the specifications have been categorized into security areas and divided into functional, non-functional, and assurance requirements.

The overall conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security

solutions. The ISO/IEC standard for security management [15] has been used as an example of how a standard could help to specify better security requirements.

3.2 Run-Time Buffer Overflow Prevention

In our paper "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention" [2] (Chapter 6) we investigate the effectiveness of four publicly available tools for run-time prevention of buffer overflow attacks—namely the GCC compiler patches StackGuard, Stack Shield, and ProPolice, and the security library Libsafe/Libverify. From an in-depth understanding of how buffer overflow attacks work we build a testbed with identified attack forms. Then the four tools are compared theoretically and empirically with the testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [5].

3.3 Compile-Time Intrusion Prevention

In our paper "A Comparison of Publicly Available Tools for Static Intrusion Prevention" [3] (Chapter 7) we investigate the effectiveness of five publicly available static intrusion prevention tools—namely the security testing tools ITS4, Flawfinder, RATS, Splint and BOON. From an in-depth understanding of how buffer overflow and format string attacks work we build a testbed with identified security bugs. All the tools are run in an empirical test with our testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [5].

3.4 More Generic Compile-Time Intrusion Prevention

The paper "Modeling and Visualizing Security Properties of Code using Dependence Graphs" [4] (Chapter 8) discusses the problem of modeling security properties, including what we call the *dual modeling problem*, and

ranking of potential vulnerabilities. The discussion is based on our own experience and the results of a brief survey of eight existing static analysis tools performing a deeper analysis than the tools tested and compared in Chapter 7.

We conclude that several categories of security properties can be statically checked but there is need of a generic solution since no programmer can be expected to run several analysis tools for the sake of security. The first step toward such a solution is to define a modeling formalism that covers all necessary aspects and allows for static analysis.

We propose dependence graphs decorated with type and range information as a more generic way of modeling security properties of code. These models can be used to characterize both good and bad programming practice as shown by our examples. They can also be used to visually explain code properties to the programmer. Finally, they can be used for pattern matching in static security analysis of code.

Chapter 4

Related Work

Each of the papers included in this thesis includes references to previous work related to the problems addressed in that particular paper. I have thus chosen to restrict this presentation of related work to more recent work in the area of run-time buffer overflow prevention where a lot of research has been done since our study in 2002, and a brief presentation of two recent studies of attack forms and prevention techniques.

4.1 Run-Time Intrusion Prevention

The *dynamic* or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks.

Run-time intrusion prevention often ends up becoming an intrusion detection system building on program and/or environment specific solutions, terminating execution in case of an attack. The techniques are often complete in the way that they can provably secure the targets they are designed

to protect and will produce no false positives (one proof can be found in a paper by Chiueh and Hsu [16]).

The general weakness of these techniques lies in the fact that they all try to solve *known* security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. Whenever an attacker has figured out a new way of exploiting a bug, these run-time solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs using the same attack method. Another potential drawback is the performance overhead produced by the run-time checks.

Many security researchers believe that run-time techniques have a better chance of solving the problems of code vulnerabilities than compile-time approaches. The most common argument we have heard is that programmers are either too lazy or have too little time to use static analysis tools and patch their code before delivery.

In Chapter 6 we compare four tools for run-time protection against buffer overflows, publicly available in 2002. Here we briefly look at the research progress since then, both in new tools and techniques, and in more recent studies of attacks and countermeasures.

The research in countering buffer overflow attacks had gone in several directions. We have identified six general categories or techniques:

- Canary-based
- Boundary checking
- Copying and checking target data
- Randomized instructions
- Library wrappers
- Non-executable memory

4.1.1 Canary-Based Tools

This technique was invented by Cowan *et al* [17] and prevents buffer overflows by adding a *canary value* to sensitive memory regions. The canary

is integrity checked before the sensitive memory is used. If the canary has been changed (i.e. *killed*) the sensitive memory may have been corrupted and some kind of alarm is raised.

To detect heap-based overflows Robertson *et al* prepend a canary value to each memory chunk structure [18]. Before the memory management information stored in a memory chunk is used the canary value is checked. If there has been a buffer overflow affecting the management information the canary has been killed. Canary values are generated at start-up and protected with `mprotect()`. The technique is very similar to StackGuard with random canaries [17].

There are new prevention schemes that reuse previous tools as part of their functionality. Sidiroglou and Keromytis have built a system for automatic worm detection and patch generation [19] where they use the canary-based *ProPolice* [20] to detect buffer overflows.

4.1.2 Boundary Checking Tools

Standard C and C++ do not have run-time bounds checking unlike modern languages such as Java. This is one of the fundamental vulnerabilities that make buffer overflow attacks possible. Researchers have implemented variants of C compilers that include boundary checking in binaries.

In 1997 Jones and Kelly presented a GCC compiler patch in which they implemented run-time bounds checking of variables [21]. For each declared storage pointer they keep an entry in a table where the base and limit of the storage is kept. Before any pointer arithmetic or pointer dereferencing is made, the base and limit is checked in the table.

Sadly their solution suffered from performance penalties of more than 400 %, as well as incompatibility with real-world programs [22]. The compatibility problem with their approach was that once a memory pointer was deemed out-of-bounds it could never be sanitized, even if pointer arithmetics made it point to the original memory again. It turned out such calculations are quite frequent in real-life code [23].

Ruwase and Lam continued Jones' and Kelly's work and have implemented a GCC patch called "CRED" [23]. Their goals were for the run-time checks to impose less overhead and provide better compatibility. To enhance performance they only perform boundary checks on string buffers

since they consider such buffers the most likely ones vulnerable to security attacks. With such a restriction most of the programs they tested suffered less than 26 % overhead. Worst case was a string intensive email program which suffered 130 % overhead.

Compatibility was solved by storing out-of-bounds pointer values in so called *out-of-bounds objects*. If pointer arithmetics using the out-of-bounds pointer results in an in-bounds address the pointer is sanitized.

4.1.3 Tools Copying and Checking Target Data

Stack Shield [24] and Libverify [25] were the first buffer overflow prevention tools that used the technique of storing copies of return addresses on a separate stack. When a function returned, it's stored return address was checked against the copy on the separate stack. If the addresses differed either the correct address was copied back or execution was halted. Stack Shield is a compiler patch whereas Libverify patched the code during load.

Nebenzahl and Wool have developed a technique for instrumenting Windows binaries at install-time with a separate stack for copies of return addresses [26]. When the program is installed they inject code for storing the current return address into all function prologues, and code for checking the return address into all function epilogues. They do not need access to the source code as opposed to Stack Shield, and they only inject code once for every binary as opposed to Libverify.

Chiueh and Hsu presented a compiler patch called *RAD* in 2001 [16]. It used a separate stack to keep copies of return addresses similar to Stack Shield. Smirnov and Chiueh have continued the work and implemented a more complex GCC patch called *DIRA* [27]. Apart from the separate stack with copies of return addresses, DIRA keeps copies of function pointer values in a special buffer. Every time a function pointer is dereferenced it is compared with the stored value.

DIRA also keeps track of memory updates at run-time and uses this information to perform a roll-back if an attack against a return address or function pointer is detected. Updates to files or local variables are not tracked and can thus not be rolled back. DIRA also does simple dataflow analysis to track external data connected to the attack. Performance overhead varies between 8 % and 60 %.

4.1.4 Tools using Randomized Instructions

Successful intrusion attacks need to change the control-flow of the process under attack. Control is redirected either to injected attack code or to already loaded code that performs what the attacker wants. By introducing randomness in how pointers and/or code are interpreted such attacks can be countermeasured.

Kc, Keromytis, and Prevelakis have investigated the use of randomized (encrypted) instruction-sets to countermeasure injected attack code [28]. Binary files, libraries etc. are XOR:ed with a single random key and before execution the instructions are decrypted. If an attacker injects his or her own code the decryption will corrupt that code and probably make the process crash. Their scheme would need a special CPU that allow such decryption of instructions. To test this they emulated such a CPU and ran a few encrypted applications. The performance overhead with the emulated CPU stretched from 34 % to 1700 %.

Barrantes, Ackley, Forrest, and Stefanovic simultaneously implemented a slightly different system for instruction-set randomization [29]. Instead of a system-wide code encryption key they generate a unique key for each program, encrypting the code during load into memory. To decrypt and run the code they use the Valgrind IA32 - to - IA32 binary translator [30].

Cowan, Beattie, Johansen, and Wagle have implemented *PointGuard*, a compiler technique to defend against buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers [31]. Pointers are safe in registers because registers are not addressable, so PointGuard depends on pointers always being loaded into registers before being dereferenced. If an attacker changes a pointer in memory it will be decrypted to an unpredictable value and thus most probably cause a crash. The random encryption key is generated at the time the process starts and is never shared outside the process address space.

4.1.5 Library Wrappers

Originally, the work with buffer overflow prevention through library wrappers was done by Baratloo, Singh, and Tsai, and their tool was called *Libsafe* [32].

It patches library functions in C that constitute potential buffer overflow vulnerabilities. In the patched functions a range check is made before the actual function call. As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address nor the old base pointer. Further protection was provided with *Libverify* using a dynamic approach with a separate return address stack [25].

Avijit, Gupta and Gupta continued the work by Baratloo *et al* by implementing *LibsafePlus* and *TIED* [33, 34]. It collects and stores information about the sizes of both stack and heap buffers. This information is used run-time to ensure that no character buffers are written past their limit.

Static buffer size is collected compile-time by exploiting debugging information produced by a specific compiler option. Dynamic buffer size information is collected run-time by interception of calls to `malloc()` and `free()`. Finally, the original technique with wrappers for dynamically linked libraries handling strings is used to check the bounds.

Their main contributions are a more precise boundary check of stack buffers than the previous solution, and boundary check of heap buffers.

4.1.6 Non-Executable and Randomized Memory

The Linux kernel patch from the Openwall Project was the first to implement a non-executable stack [35]. Not allowing execution of code stored on the stack effectively stops execution of *stack injected* attack code.

Two more recent kernel patches that deny execution both on the stack and on the heap are PaX [36] and ExecShield [37]. They also randomize address offsets to further countermeasure buffer overflow attacks. It is possible to hijack a process without injecting code, but the attacker still has to redirect control-flow to the address where the code of his or her choice starts. By randomizing address offsets this part of the attack gets very hard.

4.2 Related Studies on Attacks and Prevention

Pincus and Baker present an overview of recent advances in exploitation of buffer overflows [38]. Their main conclusion is that often heard assumptions about buffer overflows are not true—buffer overflows do not all inject code, do not all target the return address, and do not all abuse buffers on the stack. The article briefly discusses:

- Injection of attack parameters instead of attack code
- Attacks targeting function pointers, data pointers, exception handlers, and pointers to virtual function tables in C++
- Heap-based overflows

Silberman and Johnson made a presentation of buffer overflow prevention techniques at the Black Hat USA 2004 Briefings & Training. Their presentation largely coincides with what we presented in our NDSS'03 paper on buffer overflow prevention [2] (Chapter 6). They have also released a Windows port of our NDSS'03 buffer overflow testbed called "Attack Vector Test Platform" [39].

Chapter 5

Security Requirements—A Field Study of Current Practice[1]

5.1 Abstract

The number of security flaws in software is a costly problem. In 2004 more than ten new security vulnerabilities were found in commercial and open source software every day. More accurate and consistent security requirements could be a driving force towards more secure software. In a field study of eleven software projects including e-business, health care and military applications we have documented current practice in security requirements. The overall conclusion is that security requirements are poorly specified due to three things: inconsistency in the selection of requirements, inconsistency in level of detail, and almost no requirements on standard security solutions. We show how the requirements could have been enhanced by using the ISO/IEC standard for security management.

[1]Published in the Proceedings of the Symposium on Requirements Engineering for Information Security 2005 (SREIS). Authors: John Wilander and Jens Gustavsson [1].

Keywords: security requirements, non-functional requirements

5.2 Introduction

According to statistics from CERT Coordination Center, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software [7]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [8].

For consumers of software the security of the products they use relies heavily on the security requirements specified for the products. If these requirements are poorly specified there is nothing saying that the producers will strive for security. Instead, costs and time will be focused on meeting the other requirements, and security issues may be left for maintenance in the infamous *penetrate and patch* manner [9].

To build more secure software, accurate and consistent security requirements must be specified. We have investigated current practice by doing a field study of eleven requirement specifications on IT systems being built 2003 through 2005. To evaluate the outcome we have looked into documentation of security requirements from the requirements engineering community as well as from the security community. Requirements found in the specifications have been categorized into security areas and divided into functional, non-functional, and assurance requirements. The ISO/IEC standard for security management has been used as an example of how a standard could help to specify better security requirements.

The rest of this paper is organized as follows. In Section 5.3 we look at how security requirements have been defined within the requirements engineering community and the security community. Next, Section 5.4 discusses security testing to verify that security requirements have been met. Section 5.5 presents and discusses our field study of eleven requirements specifications and what they specify in terms of security. Finally, Section 5.6 concludes our work.

5.3 Security Requirements

A subgroup of software requirements is security requirements. A lot of work and research has been done to define and standardize security requirements, especially by military organizations. Here we look at (examples of) how security requirements are defined within the requirements engineering (RE) community and the security community.

5.3.1 From a RE Point of View

Within requirements engineering security is often conceived as a non-functional requirement along with such aspects as performance and reliability, and is generally considered hard to manage [48, 49, 50, 51].

There are several (partially overlapping) definitions of functional and non-functional requirements. The one used in this paper is based on the IEEE definition [52], Thayer and Thayer's glossary [53], extended by Burge and Brown [48].

Functional Requirement. A functional requirement (*FR*) defines something the system must do, capturing the nature of the interaction between the component and its environment. A FR must be *testable*, which means it is possible to demonstrate that the requirement has been met by a test case resulting in pass or fail [48, 52].

Non-Functional Requirement. A non-functional requirement (*NFR*) is a software requirement that describes not what the software will do, but how the software will do it. NFRs restrict the manner in which the system should accomplish its function. NFRs tend to be general and concern the whole system, not just some parts [48, 53].

In their paper on the future of software engineering Premkumar Devanbu and Stuart Stubblebine discuss security requirements. They define them as:

Security Requirement. A security requirement is a manifestation of a high-level organizational policy into the detailed requirements of a specific system [51].

5.3.2 From a Security Point of View

One of the seminal documents on security requirements is the *Common Criteria*, or *CC*. The CC is a standard and is meant to be used as the basis for *evaluation* of security properties of IT systems [54].

> "The CC will permit comparability between the results of independent security evaluations. It does so by providing a common set of requirements for the security functions of IT products and systems and for assurance measures applied to them during a security evaluation."

Following the CC standard, consumers of software produce a *Protection Profile* that identifies desired security properties of a product. The Protection Profile is a list of security requirements. Producers on the other hand create a *Security Target* that identifies the security-relevant properties of the software. A Security Target can meet one or more Protection Profiles. CC distinguishes between two types of security requirements—functional and assurance:

Security Functional Requirement (CC). Security functional components express security requirements intended to counter threats in the assumed operating environment. These requirements describe security properties that users can detect by direct interaction with the system (i.e. inputs, outputs) or by the system's response to stimulus.

Security Assurance Requirement (CC). Requiring assurance means requiring active investigation which is a process requirement. Active investigation is an evaluation of the IT system in order to determine its security properties.
Common Criteria lists what can be done in terms of assurance through evaluation. We highlight a few things here to give an example of what these requirements can look like:

- Analysis and checking of process(es) and procedure(s);
- checking that process(es) and procedure(s) are being applied;
- analysis of functional tests developed and the results provided;

- independent functional testing; and
- penetration testing.

Another relevant standard is the *ISO/IEC 17799 Information technology—Code of practice for information security management* [15]. The section on "Systems development and maintenance" includes ten pages specifying requirements and explaining considerations for techniques such as input validation, encryption, and security of system files.

The ISO/IEC standard does not discuss functional, non-functional, or assurance requirements as such.

5.4 Security Testing

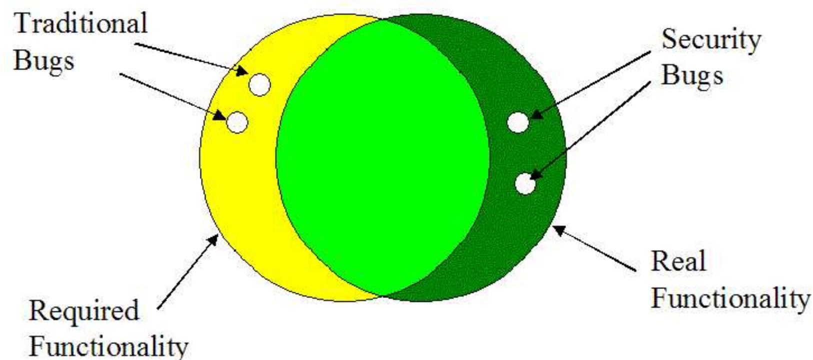


Figure 5.1: Finding security bugs through testing often means testing for side-effects and functionality outside the requirement specification.

Closely related to requirements is testing. If something is considered a requirement there needs to be some way to verify that it has been met. This can be done with testing where the outcome is pass or fail.

"Traditional" bugs are deviations from the requirement specification, either by doing B when supposed to do A, or by only doing B when supposed to do A and B.

Thompson and Whittaker write about running test cases to find security bugs [55]. Such bugs often differ from traditional bugs by being hidden in side effects. Finding security bugs means finding out what the system *also* does, apart from the specified functionality. Thompson and Whittaker's Venn diagram shows this (see Figure 5.1).

Requirements on absence of side effects are typically non-functional. Specifying what the system must not do clearly restricts in what way the functional requirements can be fulfilled. Moreover, requirements on *testing* of side effects are not only non-functional but also a kind of security assurance requirement.

This stresses that we need non-functional requirements, and specifically security assurance requirements to specify more secure systems. As we will see later such requirements are rare in current practice (see Section 5.5).

5.5 Field Study of Eleven Requirements Specifications

We have studied eleven requirements specifications of IT systems being built 2003 through 2005. In this section we first present an overview of security areas found in the specifications, and an overview of the systems and organizations that have written the specifications. Next, we present both a summarized and a detailed categorization of all security requirements found. The categorization is done into security areas and into functional, non-functional, and assurance requirements. Finally, we discuss the outcome and reflect on potential shortcomings in the material.

On an abstract level we have categorized the security requirements into well-known security areas. A full description along with examples for each category can be found in Internet security glossaries [56, 57].

5.5.1 Systems in the Field Study

In our study we have taken advantage of the fact that all requirement specifications used for public procurement by Swedish Government or local authorities are public documents. The authorities are also required by law to publish their requests for tenders, and all such requests are categorized

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Access Control/Roles	1	11	6	5	8	5	4	5	3		3
Attack Detection							2	4		3	
Backup		5	9	2			2	2			
Digital Signatures			1		1	1	1	2			1
Encryption							4	1			1
Integration							2	1			
Logging		9	3	1	11	1	5	8	1		
Login		5	3	3	8	2		2	1		2
Privacy			2							1	
Authentication							2	4	2		1
Availability	1		3			1	6	4		3	1
Design/Implementation					1			6			1
Physical Security								6			
Risk Analysis										1	
Security Management								2		2	
Security Testing								1			

Table 5.1: Overview of security requirements on eleven IT systems being built during 2003-2005. The double horizontal line divides the requirement categories into mostly functional (above) and mostly non-functional (below). Figures tell how many requirements were found in each category.

depending on the type of products or services bought. The categorization is called Common Procurement Vocabulary (CPV), which is a European standard [58].

We used a commercial database to find “Computer and related services” purchases made by Swedish Government or local authorities from January 2003 to June 2004 [59]. In Table 5.1 you find a summary of all security requirements found. Here is a brief description of the systems studied:

Billing (City of Jönköping). A billing system for drinking water, sewage,

and garbage collection.

Accounting (Cities of Dalsland). System for handling ledgers, accounting, and budgets for five cities in the province of Dalsland.

Salary/Staff 1 (The cities of Kinda, Ödeshög, Boxholm, and Ydre). System for administration of salaries and staff within the cities.

Salary/Staff 2 (The cities of Stenungsund and Tjörn) System for administration of salaries and staff within the cities.

E-Business (The cities of Skövde, Falköping, Karlsborg, Mariestad, Tibro, Tidaholm, and Hjo). System for electronic trade and business including billing.

Defense Materiel (Swedish Defence Materiel Administration). Web-based marketplace for consulting services to the Swedish Armed Forces.

Medical Advice (The Federation of County Councils). System for managing medical advice by phone on a national level. Redirection of calls, queue management, work-flow management, medical documentation, and statistics.

Health Care 1 (Stockholm County Council). Integration platform to support personal medical information following patients between various health care organizations.

Health Care 2 (The city of Lomma). System for event handling in health care including personal medical records.

Highway Tolls (The City of Stockholm's Executive Office). Equipment, systems and services for handling environmental fees for all vehicles entering the city of Stockholm.

Hazmat (Swedish Maritime Administration). Ship reporting system managing mandatory reporting of hazardous goods, arrival, departure, and generated waste in accordance with EU directives.

5.5.2 Detailed Categorization of Security Requirements

In tables 5.2, 5.3, 5.4, 5.5, and 5.6 we present the complete list of security requirements found in the specifications. The list is divided into security areas and every requirement is categorized as functional, non-functional,

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Access Control/Roles											
- per person (FR)	1	4	3	2	4	3	2		1		1
- per group (FR)		1	1	1	2	2	1	1			1
- one person many roles (FR)				1	1		1				
- file access r/w/x (FR)		6	2	1				4	2		1
- role-based GUI (FR)					1						
Attack Detection											
- intrusion detection (FR)							1	2		1	
- fraud detection (FR)										2	
- antivirus (FR)							1	2			
Backup											
- in general (FR)		1	4				1				
- automatic (FR)		3	2	1			1				
- time interval (FR)		1		1							
- durability (NFR)			2								
- data versioning (FR)								2			
- done run-time (FR)			1								

Table 5.2: Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

or security assurance (subcategory of non-functional). The numbers in the table are the number of requirements found for each subcategory. For instance the "E-Business" system has four specific requirements on access control per person (see Table 5.2).

The security areas are conventional but the categorization relies on the fact that the authors of the specifications know how the various terms differ, for instance the difference between access control, authorization and login where we have found similar requirements in all categories.

It is important to note that these are the requirements found in the

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Digital Signatures											
- in general (FR)					1	1		1			
- use of standard (NFR)								1			1
- use of PKI (FR)			1								
- for data origin (FR)							1				
Encryption											
- use of standard (NFR)							1				1
- during login (FR)							1				
- filesystem (FR)							1	1			
- network traffic (FR)							1				
Integration											
- with firewall (FR)							1				
- with anti-virus (FR)							1				
- with external PKI (FR)								1			
Logging											
- in general (FR)		6	1	1	1	1		1			
- automatic (FR)							3	3			
- what info to be logged (FR)		3	2		8			2			
- log not changeable (FR)					1		2	2	1		
- tool for log analysis (FR)					1						

Table 5.3: (Continued) Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

specifications, thus *not* a complete list of possible security requirements. For a complete list we refer to published standards such as Common Criteria [54] and ISO/IEC standard for security management [15].

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Login											
- username, password (FR)		2		1		1					1
- password change (FR)		2	1	1	2						1
- smart card (FR)								1			
- Single Sign-On (FR)			1	1	1			1	1		
- automatic logout (FR)		1	1		1	1					
- non-guessable passwords (FR)					1						
- resticted login attempts (FR)					1						
- inactivate old accounts (FR)					1						
- password re-use (FR)					1						
Privacy											
- anonymity (FR)										1	
- classification (FR)											

Table 5.4: (Continued) Detailed categorization of mostly functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional.

5.5.3 Discussion

Data from the field study show that—(1) Security requirements are poorly specified, and (2) The security requirements specified are mostly functional.

Security Requirements are Poorly Specified

To support the conclusion that the security requirements are poorly specified we highlight three things:

1. Inconsistent selection of security requirements
2. Inconsistent level of detail
3. Security standards are not required

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Authentication											
- use of standard (NFR)								3			1
- per person (NFR)							1				
- per system/entity (NFR)							1	1			
- smart card (FR)									1		
- biometrics (FR)									1		
Availability											
- 24h/day, 7 days/week (NFR)	1					1		1			1
- precentage uptime (NFR)			1					1		2	
- redundant power and net (NFR)			2				3	1			
- redundant data (NFR)							3			1	
- automatic restart (FR)								1			
Design/Implementation											
- compartmentalize (NFR)								1			
- input validation (NFR)								1			
- output validation (NFR)								1			
- referential integrity (NFR)					1			1			
- file integrity (NFR)								2			
- fault tolerant interfaces (NFR)											1

Table 5.5: Detailed categorization of mostly non-functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional, and (SAR) means security assurance (subcategory of non-functional).

Inconsistent Selection of Security Requirements. In several of the specifications studied we note that some relevant security areas are fairly well specified whereas other are completely left out. Typically, a need for security has been expressed with detailed functional security requirements whereas non-functional requirements are left out. This may lead to security problems (see Section 5.4).

	Billing	Accounting	Salary/Staff 1	Salary/Staff 2	E-Business	Defense Materiel	Medical Advice	Health Care 1	Health Care 2	Highway Tolls	Hazmat
Physical Security											
- in general (NFR)								1			
- fire (NFR)								2			
- water/moist (NFR)								1			
- physical intrusion (NFR)								2			
Risk Analysis											
- fraud risk (SAR)										1	
Security Management											
- use of ISO/IEC standard (SAR)								2		2	
Security Testing											
- availability, stress test (SAR)								1			

Table 5.6: (Continued) Detailed categorization of mostly non-functional security requirements on eleven IT systems being built during 2003-2005. (FR) means functional, (NFR) means non-functional, and (SAR) means security assurance (subcategory of non-functional).

Examples of such inconsistencies can be seen in access control/roles where all systems have requirements (two referring to standard) which indicates that restricted access is important. At the same time only three specifications require some kind of encryption of data communication and only two specifications require physical security including restricted physical access.

Inconsistent Level of Detail. Some security requirements have a high level of detail whereas others in the same specification are only specified on a general level. This might indicate that the organizations specifying the security requirements rely heavily on local competence and not standards.

We call this phenomenon *local heroes*—for instance, there might be someone who knows very much about backup systems and thus the specifications on backup become detailed and fairly complete. But in other

security areas the organization does not have an expert, which leads to under-specified requirements in that area.

This phenomenon can be seen in for instance the “E-Business” system where the requirements on logging are very detailed (eight requirements on what info to be logged) and at the same time digital signatures are specified as “The system should be able to handle the use of electronic signatures” with no further details.

In the specification of “Salary/Staff 1” we find detailed requirements on backup (automation, durability, and run-time backup), while in the same specification the lone requirement on digital signatures is “The system should handle electronic signatures and interfaces to PKI cards etc”.

Security Standards are Not Required. Many security areas have well-known and rigorously reviewed standards such as encryption and access control policies. The specifications studied very seldom require these standards to be followed. Instead the requirements specified leaves to designers and implementers to choose or even invent the technology to be used. Such an ad-hoc approach to security is known to lead to problems [9].

None of the specifications explicitly requires a standard policy for access control. In the case of digital signatures two out of six specifications explicitly require a standard solution. And for the area attack detection no publicly known system is required which means the producer can implement his/her own anti-virus software etc.

Security Requirements are Mostly Functional

As mentioned in Section 5.3.1, security is often conceived as a non-functional requirement, and as such it is known to be hard to manage. However, our study shows that in more than 75% (164 out of 216) of the cases, security requirements boil down to functional requirements. This transformation of abstract non-functional requirements into concrete functional requirements is known and resembles Chung *et al's* technique of “refining initial high-level goals to detailed concrete goals” [49].

However, the kind of non-functional security assurance requirements discussed in Section 5.4 are left out in almost all cases—we identified 6 such requirements out of 216. The security areas risk analysis, standardized security management, and security testing were categorized as secu-

rity assurance. The overall distribution of requirements is; CC's security functional requirements divided into functional (76%) and non-functional (21%), and last CC's security assurance requirements as non-functional (3%).

Security Requirements Absent

A natural question is—what security requirements are left out in the specifications studied? Since we decided to list only the requirements present in at least one specification, a comparison with a more complete list would indicate what could be gained. A fair comparison can be made in terms of level of detail. If a security requirement is specified it is unlikely that it has been deliberately under-specified.

To make such a comparison we have chosen two security areas, digital signatures and logging, and listed what the ISO/IEC standard for security management specifies. The reason for choosing this standard was that "Health Care 1" and "Highway Tolls" require that standard to be used.

In the case of the "E-Business" system the requirement on digital signatures was formulated as: "The system should be able to handle the use of electronic signatures". Reading the ISO/IEC standard we find detailed information on what to consider when requiring digital signatures:

- Protection of confidentiality of signature keys
- Protection of integrity of public key
- Quality of signature algorithm
- Bit-length of keys
- Signature keys should differ from keys for encryption
- Assure proper legal binding of the signatures

Logging is specified without standards in seven of the studied projects and specified by referral to standards in two of the projects. If we look at the seven projects with no referral to external documents, the ISO/IEC standard again provides requirements left out in the specifications:

- Separation of users logged and reviewers of the log
- Protection against de-activation
- Policy for who can change what to be logged
- Protection against logging media being exhausted

The subcategory "what info to be logged" can be further broken down into specific pieces of information. Three out of the seven projects above have specific requirements in what information to be logged. From the ISO/IEC standard we get the following list of left out requirements:

- User IDs
- Date and time of log-on and log-off
- Terminal ID and location
- Successful and rejected system access attempts and data access attempts
- Archiving of logs

5.5.4 Possible Shortcomings

There are possible shortcomings to our study. First, we want to stress that we do not have access to any kind of risk analysis documents underlying the security requirements specified. Therefore we cannot know if certain security areas have been left out because of deliberate decisions or because of lack of information or knowledge. As a consequence we do not judge the requirements as good or bad, but rather analyze the consistency and the use of standards.

Some of the requirements found in the specifications studied were hard to categorize in a clear way, mostly due to the diversity in definitions of non-functional requirements. Therefore the categorization should not in all cases be interpreted as a given fact.

Using requirement specifications made for public procurement in Sweden for our field study is a decision made primarily because of the availability of them. Commercial entities tend to have little interest in making their requirement specifications available for research. This limited scope affects the validity of the study.

5.6 Conclusions

We conclude that current practice in security requirements is poor. Our field study shows that security is mainly treated as a functional aspect composed of security features such as login, backup, and access control. Requirements on how to secure systems through assurance measures are left out. Nonetheless, all systems studied have some form of security requirements and most of them have detailed requirements at least in certain security areas. This shows that security is not neglected as such.

The RE community often conceives security as a non-functional requirement and thus generally hard to manage. Our study shows that security requirements are both functional and non-functional. In the functional case they represent abstract security features broken down into concrete functional requirements. In the non-functional case they are either restrictions on design and implementation, or requirements on assurance measures such as security testing.

Following standards and not relying on local competence would make management of security functional requirements no harder than other functional requirements. Thus security requirements being hard to manage mainly holds for security assurance requirements.

5.7 Acknowledgments

We would like to sincerely thank the reviewers in the SREIS program committee, and our previewers—David Byers and Kristian Sandahl.

Chapter 6

A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention[1]

6.1 Abstract

The size and complexity of software systems is growing, increasing the number of bugs. Many of these bugs constitute security vulnerabilities. Most common of these bugs is the buffer overflow vulnerability. In this paper we implement a testbed of 20 different buffer overflow attacks, and use it to compare four publicly available tools for dynamic intrusion prevention aiming to stop buffer overflows. The tools are compared empirically and theoretically. The best tool is effective against only 50% of the attacks and there are six attack forms which none of the tools can handle.

[1]Published in the Proceedings of the 10th Network & Distributed System Security Symposium 2003 (NDSS), 2003. Authors: John Wilander and Mariam Kamkar [2].

Keywords: security intrusion; buffer overflow; intrusion prevention; dynamic analysis

6.2 Introduction

The size and complexity of software systems is growing, increasing the number of bugs. According to statistics from Coordination Center at Carnegie Mellon University, CERT, the number of reported vulnerabilities in software has increased with nearly 500% in two years [60] as shown in figure 6.1.

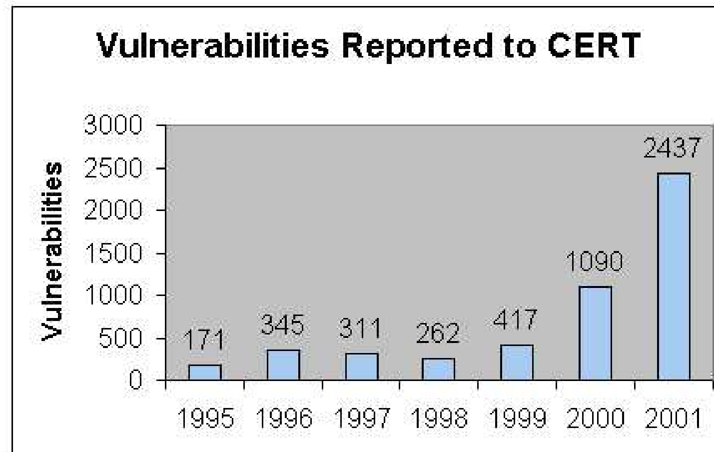


Figure 6.1: Software vulnerabilities reported to CERT 1995–2001.

Now there is good news and bad news. The good news is that there is lots of information available on how these security vulnerabilities occur, how the attacks against them work, and most importantly how they can be avoided. The bad news is that this information apparently does not lead to fewer vulnerabilities. The same mistakes are made over and over again which, for instance, is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows stand for about 50% of the vulnerabilities reported by CERT [61].

In the middle of January 2002 the discussion about responsibility for security intrusions took a new turn. The US National Academies released a prepublication recommending that policy-makers create laws that would hold companies accountable for security breaches resulting from vulnerable products [10], which received global media attention [11, 12]. So far, only the intruder can be charged in court. In the future, software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

One starting point would, or could be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most likely be more effective, but given the amount of existing software ("legacy code"), the desire for modular design using software components programmed earlier, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools that aim to clean up vulnerable source code are necessary. A further discussion of this issue can be found in the January/February 2002 issue of IEEE Software [14].

In this paper we investigate the effectiveness of four publicly available tools for dynamic prevention of buffer overflow attacks–namely the GCC compiler patches StackGuard, Stack Shield, and ProPolice, and the security library Libsafe/Libverify. Our approach has been to first develop an in-depth understanding of how buffer overflow attacks work and from this knowledge build a testbed with all the identified attack forms. Then the four tools are compared theoretically and empirically with the testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [5].

6.2.1 Scope

We have tested publicly available tools for run-time prevention of buffer overflow attacks. The tools all apply to C source code, but using them requires no modifications of the source code. We do not consider approaches

that use system specific features, modified kernels, or require the user to install separate run-time security components. The twenty buffer overflows represent a sample of the potential instances of buffer overflow attacks and not on the likelihood of a specific attack using the sample instance.

6.2.2 Paper Overview

The rest of the paper is organized as follows. Section 6.3 describes process memory management in UNIX and how buffer overflow attacks work. Section 6.4 presents the concept of intrusion prevention and describes the techniques used in the four analyzed tools. Section 6.5 defines our testbed of twenty attack forms and presents our theoretical and empirical comparison of the tools' effectiveness against the previously described attack forms. Section 6.6 describes the common shortcomings of current dynamic intrusion prevention. Finally sections 6.7 and 6.8 present related work and our conclusions.

6.3 Attack Methods

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according to definitions in, for example, the Internet Security Glossary [57]. In our context an intrusion or a successful attack aims to *change the flow of control*, letting the attacker execute arbitrary code. We consider this class of vulnerabilities the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation.

6.3.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function that writes to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes of space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing the flow of control occurs by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular target is the return address on the stack. But programmer defined *function pointers* and so called *longjmp buffers* are equally effective targets of attack.

6.3.2 Memory Layout in UNIX

To get a picture of the memory layout of processes in UNIX we can look at two simplified models (for a complete description see "Memory Layout in Program Execution" by Frederick Giasson [62]). Each process has a (partial) memory layout as in the figure below:

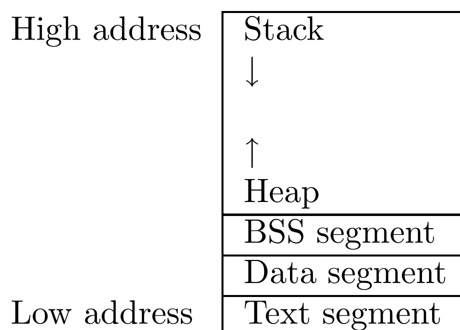


Figure 6.2: Memory layout of a UNIX process.

The machine code is stored in the text segment and constants, arguments, and variables defined by the programmer are stored in the other memory areas. A small C-program shows this (the comments show where each piece of data is stored in process memory):

```
static int GLOBAL_CONST = 1;      // Data segment
static int global_var;            // BSS segment

// argc & argv on stack, local
int main(argc **argv[]) {
  int local_dynamic_var;          // Stack
  static int local_static_var;    // BSS segment
  int *buf_ptr=(int *)malloc(32); // Heap
  ... }
```

For each function call a new *stack frame* is set up on top of the stack. It contains the return address, the calling function's base pointer, locally declared variables, and more. When the function ends, the return address instructs the processor where to continue executing code and the stored base pointer gives the offset for the stack frame to use.

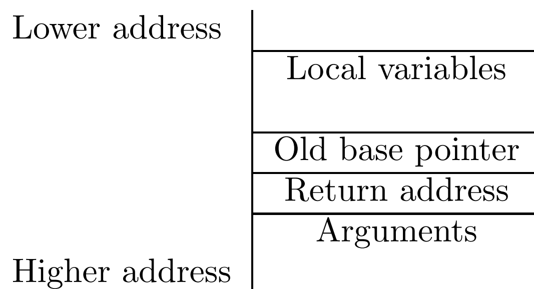


Figure 6.3: The UNIX stack frame.

6.3.3 Attack Targets

As stated above the target for a successful change of control flow is a code pointer. There are three types of code pointers to attack [63]. But Hiroaki Etoh and Kunikazu Yoda propose using the old base pointer as an attack target [20]. We have implemented their proposed attack form and proven

that the old base pointer is just as dangerous a target as the return address (see section 6.3.4 and 6.5). So we have four attack targets:

1. The return address, allocated on the stack.
2. The old base pointer, allocated on the stack.
3. Function pointers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.
4. Longjmp buffers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.

A function pointer in C is declared as `int (*func_ptr) (char)`, in this example a pointer to a function taking a `char` as input and returns an `int`. It points to executable code.

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Let's say function A first calls `setjmp()`, then calls function B which in turn calls function C. If C now calls `longjmp()` the control is directly transferred back to function A, popping both C's and B's stack frames of the stack.

6.3.4 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [61, 64] and have been extensively analyzed and described in several papers and on-line documents [65, 66, 67, 68]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control either by directly overflowing the code pointer or by first overflowing another pointer and redirect that pointer to the code pointer.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with

'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

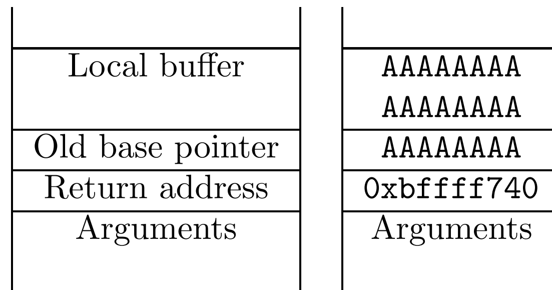


Figure 6.4: A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

The second attack target, the old base pointer, can be abused by building a fake stack frame with a return address pointing to attack code and then overflow the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.

The third attack target is function pointers. If the function pointer is redirected to the attack code the attack will be executed when the function pointer is used.

The fourth and last attack target is longjmp buffers. They contain the environment data required to resume execution from the point `setjmp()` was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code the attack will be executed when `longjmp()` is called.

Combining all these buffer overflow techniques, locations in memory and attack targets leaves us with no less than twenty attack forms. They are all listed in section 6.5 and constitute our testbed for testing of the intrusion prevention tools.

6.4 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [69] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention*. In this section we will first describe the differences between these two categories. Secondly, we describe four publicly available tools for dynamic intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against the intrusions and vulnerabilities described in section 6.3.4. This is not a complete survey of intrusion prevention techniques, rather a subset with the following constraints:

- Techniques used in the implementation phase of the software.
- Techniques that require no altering of source code to disarm security vulnerabilities.
- Techniques that are generic, implemented and publicly available, not prototypes or system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

6.4.1 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [40] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible targets. Static intrusion prevention removes the attacker's method

of entry, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database of programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued.

6.4.2 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks.

Dynamic intrusion prevention, as we will see, often ends up becoming an intrusion detection system building on program and/or environment specific solutions, terminating execution in case of an attack. The techniques are often complete in the way that they can provably secure the targets they are designed to protect (one proof can be found in a paper by Chiueh and Hsu [16]) and will produce no false positives. Their general weakness lies in the fact that they all try to solve *known* security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. Whenever an attacker has figured out a new way of exploiting a bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs using the same attack method.

6.4.3 StackGuard

The *StackGuard* compiler invented and implemented by Crispin Cowan et al [17] is perhaps the most well referenced of the current dynamic intrusion prevention techniques. It is designed for detecting and stopping stack-based buffer overflows targeting the return address.

The StackGuard Concept

The key idea behind StackGuard is that buffer overflow attacks overwrite everything on their way towards their target. In the case of a buffer overflow on the stack targeting the return address, the attacker has to fill the buffer, then overwrite any other local variables below (i.e. on higher stack addresses), then overwrite the old base pointer until it finally reaches the return address. If we place a dummy value in between the return address and the stack data above, and then check whether this value has been overwritten or not before we allow the return address to be used, we could detect this kind of attack and possibly prevent it. The inventors have chosen to call this dummy value the *canary*.

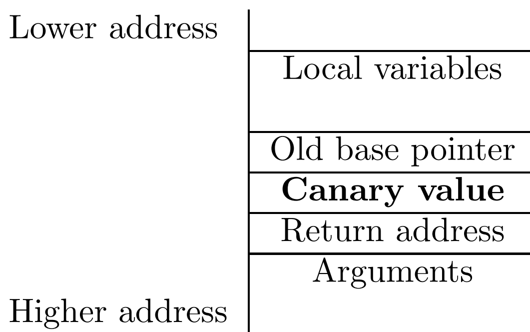


Figure 6.5: The StackGuard stack frame.

A potentially successful attack against such a system would be to somehow leave the canary intact while changing the return address, either by overwriting the canary with its correct value and thus not changing it, or by overwriting the return address through a pointer, not touching the canary. To solve the first problem, two canary versions have been suggested—firstly the *random canary* which consists of a random 32-bit value calculated at run-time, and secondly the *terminator canary* which consists of all four kinds of string termination sequences, namely `Null`, `Carriage Return`, `-1` and `Line Feed`. In the random canary case the attacker has to guess, or somehow retrieve, the random value at run-time. In the terminator canary case the attacker has to input all the termination sequences to keep the canary intact during the overflow. This is not possible since the string

function receiving the input will terminate on one of the sequences.

Note that these techniques only stop overflow attacks that overwrite everything along the stack, not general attacks against the return address. The attacker can still abuse a pointer, making it point at the return address and writing a new address to that memory position. This shortcoming of StackGuard was discovered by Mariusz Woloszyn, alias "Emsi" and presented by Bulba and Kil3er [70]. The StackGuard team has addressed this problem by not only saving the canary value but the `XOR` of the canary and the correct return address. In this way an abused return address with an intact canary preceding it would still be detected since the `XOR` of the canary and the return address has changed. If the `XOR` scheme is used the canary has to be random since the terminator canary `XOR`ed with an address would not terminate strings anymore.

Random Canaries Unsupported

While testing StackGuard we noticed that the compiler did not respond to the flag set for random canary. We e-mailed Crispin Cowan and according to him: "There is only one threat that the XOR canary defeats, and the terminator canary does not: Emsi's attack. However, if you have a vulnerability that enables you to deploy Emsi's attack, then you have many other targets to attack besides function return address values. Therefore, we dropped support for random canaries [71]". We agree that the return address is not the only attack target but it is the most popular and unlike function pointers and longjmp buffers, the return address is always present. According to Cowan's e-mail and a WireX paper a better solution is on its way called *PointGuard* which will protect the integrity of pointers in general with the same kind of canary solution [63]. This implies that PointGuard will protect against all attack forms overflowing pointers (See attack forms 3a–f and 4a–f in section 6.5).

StackGuard is available for download at `http://www.immunix.org/`.

6.4.4 Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [24]. In the current version 0.7 it implements three types of protection, two against

overwriting of the return address (both can be used at the same time) and one against overwriting of function pointers.

Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for Stack Shield. It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the return address on the normal stack is replaced by the copy on the Global Ret Stack. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the Global Ret Stack. This means only prevention and no detection of an attack. The Global Ret Stack has by default 256 entries which limits the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

Ret Range Check

A somewhat simpler but faster version of Stack Shield's protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted. Note that the Ret Range Check can detect an attack as opposed to the Global Ret Stack described above.

Protection of Function Pointers

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process' memory. That's where the programmer is likely to have implemented the functions to point at. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code

injected into the process, since injection of data only can be done into the data segment, the BSS segment, the heap, or the stack.

Stack Shield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated. This protection will give false positives if the programmer has intended to use dynamically allocated function pointers.

Stack Shield is available for download at `http://www.angelfire.com/-sk/stackshield/`.

6.4.5 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented the perhaps most sophisticated compiler protection called *ProPolice* [20].

The ProPolice Concept

Etoh's and Yoda's GCC patch ProPolice borrows the main idea from StackGuard (see section 6.4.3)—they use canary values to detect attacks on the stack. The novelty is the protection of stack allocated variables by rearranging the local variables so that `char` buffers always are allocated at the bottom, next to the old base pointer, where they cannot be overflown to harm any other local variables.

Building a Safe Stack Frame

After a program has been compiled with ProPolice the stack frame of functions look like that shown in figure 6.6.

No matter in what order local variables, pointers, and buffers are declared by the programmer, they are rearranged in stack memory to reflect the structure shown above. In this way we know that local `char` buffers can only be overflown to harm each other, the old base pointer and below. No variables can be attacked unless they are part of a `char` buffer. And

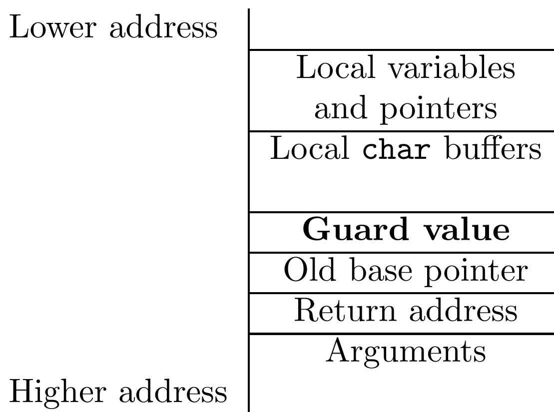


Figure 6.6: The ProPolice stack frame.

Function	Vulnerability
`strcpy(char *dest, const char *src)`	May overflow `dest`
`strcat(char *dest, const char *src)`	May overflow `dest`
`getwd(char *buf)`	May overflow `buf`
`gets(char *s)`	May overflow `s`
`[vf]scanf(const char *format, ...)`	May overflow arguments
`realpath(char *path, char resolved_path[])`	May overflow `path`
`[v]sprintf(char *str, const char *format, ...)`	May overflow `str`

Table 6.1: Vulnerable C functions that Libsafe adds protection to.

by placing the canary which they call the *guard* between these buffers and the old base pointer all attacks outside the `char` buffer segment will be detected. When an attack is detected the process is terminated.

When testing ProPolice we noticed some irregularities in when and was not the buffer overflow protection was included. It seems like small char buffers (e.g. 5 bytes) confuse ProPolice, causing it to skip the protection even if the user has set the protector flag. This gives the overall impression maybe that ProPolice is somewhat unstable.

ProPolice is available for download at `http://www.trl.ibm.com/-projects/security/ssp/`.

6.4.6 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [32] is *Libsafe*. This tool actually provides a combination of static and dynamic intrusion prevention. Statically it patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten. Further protection has been provided [25] with *Libverify* using a similar dynamic approach to StackGuard (see Section 6.4.3).

Libsafe

The key idea behind Libsafe is to estimate a safe boundary for buffers on the stack at run-time and then check this boundary before any vulnerable function is allowed to write to the buffer. Vulnerable functions they consider to be the ones in table 6.1 below.

As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address.

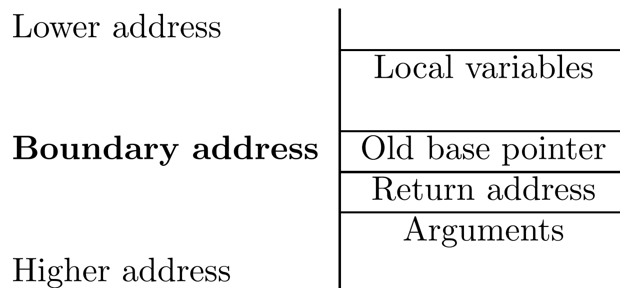


Figure 6.7: The Libsafe stack frame.

This boundary is enforced by overloading the functions in table 6.1 with wrapping functions. These wrappers first compute the length of the input as well as the allowed buffer size (i.e. from the buffer's starting point to the old base pointer) and then performs a boundary check. If the input is within the boundary the original functionality is carried out. If not the

wrapper writes an alert to the system's log file and then halts the program. Observe that overflows within the local variables on the stack, such as function pointers, are not stopped.

Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackGuard. But since this is a library it does not require recompilation of the software. As with Libsafe the library is pre-loaded and linked to any program running on the system.

The key idea behind Libverify is to alter all functions in a process so that the first thing done in every function is to copy the return address onto a *canary stack* located on the heap, and the last thing done before returning is to verify the return address by comparing it with the address saved on the canary stack. If the return address is still correct the process is allowed to continue executing. But if the return address does not match the saved copy, execution is halted and a security alert is raised. Libverify does not protect the integrity of the canary stack. They propose protecting it with `mprotect()` as in RAD (see section 6.4.7) but as in the RAD case this will most probably impose a very serious performance penalty [16].

To be able to do this, Libverify has to rearrange the code quite a bit. First each function is copied whole to the heap (requires executable heap) where it can be altered. Then the saving and verifying of the return address is injected into each function by overwriting the first instruction with a call to `wrapper_entry` and all return instructions with a call to `wrapper_exit`. The need for copying the code to the heap is due to the Intel CPU architecture. On other platforms this could be solved without copying the code [25].

Libverify is needed to give a more complete protection of the return address since Libsafe only addresses standard C library functions (as pointed out by Istvan Simon [72]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

Libsafe and Libverify are available for download at `http://www.research.avayalabs.com/project/libsafe/`.

6.4.7 Other Dynamic Solutions

The dynamic intrusion prevention techniques presented above are not the only ones. Other researchers have had similar ideas and implemented alternatives.

Tzi-cker Chiueh and Fu-Hau Hsu from State University of New York at Stony Brook have presented a compiler patch for protection of the return address [16]. They call their GCC patch *Return Address Defender*, or *RAD* for short. The key idea behind RAD is quite similar to the return address protection of Stack Shield described in Section 6.4.4. Every time a function call is made and a new stack frame is created, RAD stores a copy of the new return address. When a function returns, the return address about to be dereferenced is first checked against its copy. RAD is not publicly available.

The GCC patch *StackGhost* [73] by Mike Frantzen and Mike Shuey makes use of system specific features of the Sun Sparc Station to implement a sophisticated protection of the return address. They propose both XORing a random value with the return address (as StackGuard) as well as keeping a separate return address stack (as Stack Shield, RAD and Libverify). They also suggest using cryptographic methods instead of XOR to enhance security.

CCured and Cyclone are two recent research projects aiming to significantly enhance type and bounds checking in C. They both use a combination of static analysis and run-time checks.

CCured [74, 75] is an extension of the C programming language that distinguishes between various kinds of pointers depending on their usage. The purpose of this distinction is to be able to prevent improper usage of pointers and thus to guarantee that programs do not access memory areas they shouldn't access. CCured will change C programs slightly so that they are type safe. CCured does not change code that does not use pointers or arrays.

Cyclone [76] is a C dialect that prevents safety violations such as buffer overflows, dangling pointers, and format string attacks by ruling out certain parts of ANSI C and replacing them with safer versions. For instance `setjmp()` and `longjmp()` are unsupported (in some cases exceptions are used instead). Also pointer arithmetic is restricted. An average of 10%

of the lines of code have to be changed when porting programs from C to Cyclone.

Richard Jones and Paul Kelly 1997 presented a GCC compiler patch in which they implemented run-time bounds checking of variables [21]. For each declared storage pointer they keep an entry in a table where the base and limit of the storage is kept. Before any pointer arithmetic or pointer dereferencing is made, the base and limit is checked in the table. While not explicitly aimed for security, this technique would effectively stop all kinds of buffer overflow attacks. Sadly their solution suffered both from performance penalties of more than 400 %, as well as incompatibilities with real-world programs (according to Crispin Cowan et al [22]). Because of the bad performance and compatibility we considered Jones' and Kelly's solution less interesting for software development and excluded it from our test.

It is also possible to have support for dynamic intrusion prevention in the operating system. A popular idea is the non-executable stack. This would make injection of attack code into the stack useless. But there are many ways around this protection. A few examples include using code already linked into the program from libraries (for instance calling `system()` with the parameter `"/bin/sh"`), injecting the attack code into other memory structures such as environment variables, or by exploiting buffer overflows on the heap or in the BSS/data segment. The Linux kernel patch from the Openwall Project is publicly available and implements a non-executable stack as well as protection against attacks using library functions [35]. Since it is a kernel patch it is up to the user and not the producer of software to install it. Therefore we did not include it in our test.

David Wagner and Drew Dean have presented an interesting approach for intrusion detection that relates to the functionality of the tools described in this paper [77]. They model the program's correct execution behavior via static analysis of the source code, building up callgraphs or even equivalent context-free languages defining the set of possible system call traces. Then these models are used for run-time monitoring of execution. Any deviation from the defined 'good' behavior will make the model enter an unaccepting state and trigger the intrusion alarm. As the metric for precision in intrusion detection they propose the branching factor of the

Development Tool	Attacks prevented	Attacks halted	Attacks missed	Abnormal behavior
StackGuard Terminator Canary	0 (0%)	3 (15%)	16 (80%)	1 (5%)
Stack Shield Global Ret Stack	5 (25%)	0 (0%)	14 (70%)	1 (5%)
Stack Shield Range Ret Check	0 (0%)	0 (0%)	17 (85%)	3 (15%)
Stack Shield Global & Range	6 (30%)	0 (0%)	14 (70%)	0 (0%)
ProPolice	8 (40%)	2 (10%)	9 (45%)	1 (5%)
Libsafe and Libverify	0 (0%)	4 (20%)	15 (75%)	1 (5%)

Table 6.2: Empirical test of dynamic intrusion prevention tools. 20 attack forms tested. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

model. A low branching factor means that the attacker has few choices of what to do next if he or she wants to evade detection.

6.5 Comparison of the Tools

Here we define our testbed of twenty buffer overflow attack forms and then present the outcome of our empirical and theoretical comparison of the tools from section 6.4.2.

We define an attack form as a combination of a technique, a location, and an attack target. As described in section 6.3.3 we have identified two techniques, two types of location and four attack targets:

Techniques. Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations. The types of location for the buffer overflow are the stack or the heap/BSS/data segment.

Attack Targets. We have four targets—the return address, the old base pointer, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Considering all practically possible combinations gives us the twenty attack forms listed below.

1. Buffer overflow on the stack all the way to the target:
   (a) Return address
   (b) Old base pointer
   (c) Function pointer as local variable
   (d) Function pointer as parameter
   (e) Longjmp buffer as local variable
   (f) Longjmp buffer as function parameter
2. Buffer overflow on the heap/BSS/data all the way to the target:
   (a) Function pointer
   (b) Longjmp buffer
3. Buffer overflow of a pointer on the stack and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter
4. Buffer overflow of a pointer on the heap/BSS/data and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable

Development Tool	Attacks prevented	Attacks halted	Attacks missed
StackGuard Terminator Canary	0 (0%)	4 (20%)	16 (80%)
StackGuard Random XOR Canary	0 (0%)	6 (30%)	14 (70%)
Stack Shield Global Ret Stack	6 (30%)	7 (35%)	7 (35%)
Stack Shield Range Ret Check	0 (0%)	10 (50%)	10 (50%)
Stack Shield Global & Range	6 (30%)	7 (35%)	7 (35%)
ProPolice	8 (40%)	3 (15%)	9 (45%)
Libsafe and Libverify	0 (0%)	6 (30%)	14 (70%)

Table 6.3: Theoretical comparison of dynamic intrusion prevention tools. 20 attack forms used. "Prevented" means that the process execution is unharmed. "Halted" means that the attack is detected but the process is terminated.

(f) Longjmp buffer as function parameter

Note that we do not consider differences in the likelihood of certain attack forms being possible, nor current statistics on which attack forms are most popular. However, we have observed that most of the dynamic intrusion prevention tools focus on the protection of the return address. Bulba and Kil3r did not present any real-life examples of their attack forms that defeated StackGuard and Stack Shield. Also the Immunix operating system (Linux hardened with StackGuard and more) came in second place at the Defcon "Capture the Flag" competition where nearly 100 crackers and security experts tried to compromise the competing systems [78]. This implies that the tools presented here are effective against many of the currently used attack forms. The question is: will this change as soon as this kind of protection is wide spread?

Also worth noting is that just because an attack form is prevented or halted does not mean that the very same buffer overflow can not be abused in another attack form. All of these attack forms have been implemented on the Linux platform and the source code is available from our homepage: `http://www.ida.liu.se/~johwi`.

To set up the test, the source code was compiled with StackGuard, Stack Shield, or ProPolice, or linked with Libsafe/Libverify. The overall

results are shown in table 6.2. We also made a theoretical comparison to investigate the potential of the ideas and concepts used in the tools. The overall results of the theoretical analysis are shown in table 6.3. For details of the tests see appendix .1 and .2.

Most interesting in the overall test results is that the most effective tool, namely ProPolice, is able to prevent only 50% of the attack forms. Buffer overflows on the heap/BSS/data targeting function pointers or longjmp buffers are not prevented or halted by any of the tools, which means that a combination of all techniques built into one tool would still miss 30% of the attack forms.

This however does not comply with the result from the theoretical comparison. Stack Shield was not able to protect function pointers as stated by Vendicator. Another difference is the abnormal behavior of StackGuard and Stack Shield when confronted with a fake stack frame in the BSS segment.

These poor results are all evidence of the weakness in dynamic intrusion prevention discussed in section 6.4.2, the tested tools all aim to protect *known* attack targets. The return address has been a popular target and therefore all tools are fairly effective in protecting it.

Worth noting is that StackGuard halts attacks against the old base pointer although that was not mentioned as an explicit design goal.

Only ProPolice and Stack Shield offer real intrusion prevention—the other tools are more or less intrusion detection systems. But still the general behavior of all these tools is termination of process execution during attack.

6.6 Common Shortcomings

There are several shortcomings worth discussing. We have identified four generic problems worth highlighting, especially when considering future research in this area.

6.6.1 Denial of Service Attacks

Since three out of four tools terminate execution upon detecting an attack they actually offer more of intrusion detection than intrusion prevention.

More important is that the vulnerabilities still allow for Denial of Service attacks. Terminating a web service process is a common goal in security attacks. Process termination results in a much less serious attack but will still be a security issue.

6.6.2 Storage Protection

Canaries or separate return address stacks have to be protected from attacks. If the canary template or the stored copy of the return address can be tampered with, the protection is fooled. Only StackGuard with the terminator canary offers protection in this sense. The other tools have no protection implemented and the performance penalty of such protection can be very serious—up to 200 times [16].

6.6.3 Recompilation of Code

The three compiler patches have the common shortcoming of demanding recompilation of all code to provide protection. For software vendors shipping new products this is a natural thing but for running operating systems and legacy systems this is a serious drawback. Libsafe/Libverify offers a much more convenient solution in this sense. The StackGuard and ProPolice teams have addressed this issue by offering protected versions of Linux and FreeBSD.

6.6.4 Limited Nesting Depth

When keeping a separate stack with copies of return addresses, the nesting depth of the process is limited. Only Vendicator, author of Stack Shield, discusses this issue but offers no real solution to the problem.

6.7 Related Work

Three other studies of defenses against buffer overflow attacks have been made.

In late 2000 Crispin Cowan et al published their paper "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" [63]. They

implicitly discuss several of our attack forms but leave out the old base pointer as an attack target. Comparison of defenses is broader considering also operating system patches, choice of programming language and code auditing but there is only a theoretical analysis, no comparative testing is done. Also the only dynamic tools discussed are their own StackGuard and their forthcoming PointGuard.

Only a month later Istvan Simon published his paper “A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks” [72]. It discusses pros and cons with operating system patches, StackGuard, Libsafe, and similar solutions. The major drawback in his analysis is the lack of categorization of buffer overflow attack forms (only three of our attack forms are explicitly mentioned) and any structured comparison of the tool’s effectiveness. No testing is done.

In March 2002 Pierre-Alain Fayolle and Vincent Glaume published their lengthy report “A Buffer Overflow Study, Attacks & Defenses” [79]. They describe and compare Libsafe with a non-executable stack and an intrusion detection system. Tests are performed for two of our twenty attack forms. No proper categorization of buffer overflow attack forms is made or used for testing.

6.8 Conclusions

There are several run-time techniques for stopping the most common of security intrusion attack—the buffer overflow. But we have shown that none of these can handle the diverse forms of attacks known today. In practice at best 40% of the attack forms were prevented and another 10% detected and halted, leaving 50% of the attacks still at large. Combining all the techniques in theory would still leave us with nearly a third of the attack forms missed. In our opinion this is due to the general weakness of the dynamic intrusion prevention solution—the tools all aim at protecting *known* attack targets, not all targets. Nevertheless these tools and the ideas they are built on are effective against many security attacks that harm software users today.

6.9 Acknowledgments

We are grateful to the readers who have previewed and improved our paper, especially Crispin Cowan.

Chapter 7

A Comparison of Publicly Available Tools for Static Intrusion Prevention[1]

7.1 Abstract

The size and complexity of today's software systems is growing, increasing the number of bugs and thus the possibility of security vulnerabilities. Two common attacks against such vulnerabilities are buffer overflow and format string attacks. In this paper we implement a testbed of 44 function calls in C to empirically compare five publicly available tools for static analysis aiming to stop these attacks. The results show very high rates of false positives for the tools building on lexical analysis and very low rates of true positives for the tools building on syntactical and semantical analysis.
. . .
Keywords: security intrusions, intrusion prevention, static analysis, security testing, buffer overflow, format string attack

[1]Published in the Proceedings of the 7th Nordic Workshop on Secure IT Systems, 2002. Authors: John Wilander and Mariam Kamkar [3].

7.2 Introduction

As our software systems are growing larger and more complex the amount of bugs increase. Many of these bugs constitute security vulnerabilities. According to statistics from CERT Coordination Center at Carnegie Mellon University the number of reported security vulnerabilities in software has increased with nearly 500% in two years [60].

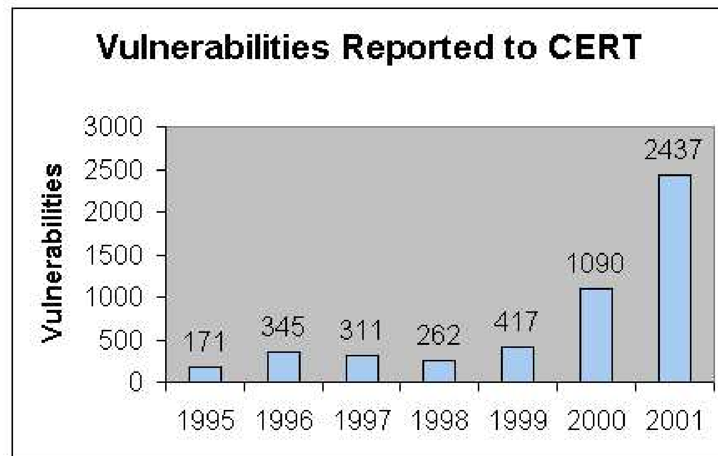


Figure 7.1: Software vulnerabilities reported to CERT 1995–2001.

Now there is good news and bad news. The good news is that there is lots of information out there on how these security vulnerabilities occur, how the attacks against them work and most importantly how they can be avoided. The bad news is that this information apparently does not lead to less vulnerabilities. The same mistakes are made over and over again which for instance is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows alone stand for about 50% of the vulnerabilities reported by CERT [61]. Equally dangerous is the *format string* vulnerability which was publicly unknown until 2000.

In the middle of January 2002 the discussion about responsibility for security intrusions took an interesting turn. The US National Academies released a prepublication recommending policy-makers to create laws that

would hold companies accountable for security breaches resulting from vulnerable products [10] which got global media attention [11, 12]. So far, only the intruder can be charged in court. In the future software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

A good starting point would be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most probably be more effective, but given the amount of existing software, the strive for modular design reusing software components, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools trying to clean up vulnerable source code are necessary. A further discussion on this issue can be found in the January/February 2002 issue of IEEE Software [14].

In this paper we investigate the effectiveness of five publicly available static intrusion prevention tools—namely the security testing tools ITS4, Flawfinder, RATS, Splint and BOON. Our approach has been to first get an in-depth understanding of how buffer overflow and format string attacks work and from this knowledge build up a testbed with identified security bugs. We then make an empirical test with our testbed. This work is a follow-up of John Wilander's Master's Thesis [5].

The rest of the paper is organized as follows. Section 7.3 describes process memory management in UNIX and how buffer overflow and format string attacks work. Here we define our testbed of 23 vulnerable functions in C. Section 7.4 presents the concept of intrusion prevention and describes the techniques used in the five analyzed tools. Section 7.5 presents our empirical comparison of the tools' effectiveness against the previously described vulnerabilities. Related work is presented in section 7.6. Finally section 7.7 contains our conclusions.

7.3 Attacks and Vulnerabilities

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according definitions in for example the Internet Security Glossary [57]. In our context an intrusion or a successful attack aims for *changing the flow of control*, letting the attacker execute arbitrary code. Software security bugs, or *vulnerabilities*, allowing these kind of intrusions are considered the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in his or her new shell, leaving the whole system open for any kind of manipulation.

7.3.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function writing to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing flow of control is made by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular code pointer

to target is the return address on the stack. But programmer defined *function pointers*, so called *longjmp buffers*, and the old *base pointer* are equally effective targets of attack.

7.3.2 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [61, 64] and has been extensively analyzed and described in several papers and on-line documents [65, 66, 68, 67]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

Local buffer	`AAAAAAAA` `AAAAAAAA`
Old base pointer	`AAAAAAAA`
Return address	`0xbffff740`
Arguments	Arguments

Figure 7.2: A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

7.3.3 Buffer Overflow Vulnerabilities

So how come there is no check whether the data fits into the destination buffer? The problem is that several of ANSI C's standard library functions rely on the programmer to do the checking, which they often do not. Many of these functions are powerful for handling strings and thus popular. More secure versions have in some cases been implemented but are not always known by programmers. There are lists of these dangerous C functions often involved in published buffer overflows [80, 41, 9]. From these lists we have chosen to take the fifteen functions considered most risky into our testbed:

1. `gets()`
2. `cuserid()`
3. `scanf()`
4. `fscanf()`
5. `sscanf()`
6. `vscanf()`
7. `vsscanf()`
8. `vfscanf()`
9. `sprintf()`
10. `strcat()`
11. `strcpy()`
12. `streadd()`
13. `strecpy()`
14. `vsprintf()`
15. `strtrns()`

This list is not exhaustive but should provide useful test data for our comparison of the tools.

7.3.4 Format String Attacks

22nd of June 2000 the first *format string attack* was published [81]. Comments in the exploit source code dates to the 15th of October 1999. Until then this whole category of security bugs was publicly unknown. Since then format string attacks have been acknowledged for being as dangerous as buffer overflow attacks. They are described in an extensive article by Team Teso [82] and also in a shorter article by Tim Newsham [83].

String functions in ANSI C often handle so called *format strings*. They allow for dynamic composition or formatting of strings using *conversion specifications* starting with the character `%` and ending with a conversion specifier. Each conversion specification results in fetching zero or more subsequent arguments.

Let's say a part of a program looks like this:

```
void print_function_1(char *string) {
  printf("%s", string); }
```

A call to `print_func_1()` would print the string argument passed. The same functionality could (seemingly) be achieved with somewhat simpler code:

```
void print_function_2(char *string) {
  printf(string); }
```

Using the function argument `string` directly will still print the argument passed to `print_function_2()`. But what if we call `print_function_2()` with a string containing conversion specifications, for example `print_function_2("%d%d%d%d")`? Then `printf()` will interpret the string as a format string and in this case assume that there are four integers stored on the stack and thus pop four times four bytes of stack memory and print the values stored there. So if programmers take this shortcut when using format string functions, the possibility arises for an attacker to inject conversion specifications that will be evaluated.

Now, considering the conversion specifier `%n` things get dangerous. `%n` will cause the format string function to pop four bytes of the stack and use that value as a memory pointer for storing the number of characters so far in the format string (i.e. the number of characters before `%n`.). So by injecting a format string containing `%n` an attacker can *write* data into the process' memory.

If an attacker is able to provide the format string to a an ANSI C format function in part or as a whole a format string vulnerability is present. By combining the various conversion specifications and making use of the fact that the format string itself is stored on the stack we can view and write on arbitrary memory addresses.

7.3.5 Format String Vulnerabilities

While the `scanf()`-family is involved in numerous of buffer overflow exploits [32] the format string attacks published concern the `printf()`-family of format string functions [82, 84]. For that reason our test only concerns the latter subset of the ANSI C format functions. So we add another eight

function calls to our testbed (`sprintf()` and `vsprintf()` are used differently here than in the buffer overflow case):

16.	`printf()`	20.	`vprintf()`
17.	`fprintf()`	21.	`vfprintf()`
18.	`sprintf()`	22.	`vsprintf()`
19.	`snprintf()`	23.	`vsnprintf()`

7.4 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [69] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention*. In this section we will first describe the differences between these two categories. Secondly, we describe five publicly available tools for static intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against vulnerabilities described in section 7.3.2. This is not a complete survey of static intrusion prevention tools, rather a subset with the following constraints:

- Tools used in the testing phase of the software.
- Tools that require no altering of source code to detect security vulnerabilities.
- Tools that are implemented and publicly available, not system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

7.4.1 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks. Their general weakness lies in the fact that the protection schemes all depend on how bugs are known to be exploited today, but they do not get rid of the actual bugs. Whenever an attacker has figured out a new attack target reachable with the same security bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs aiming for the same target.

7.4.2 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [40] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible security targets in the software. Static intrusion prevention removes the attackers tools, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued. In this paper we have chosen to focus on five publicly available tools for static intrusion prevention.

7.4.3 ITS4

In late 2000 researchers at Reliable Software Technologies, now Cigital, presented a static analysis tool for detecting security vulnerabilities in C and C++ code—*It's the Software Stupid! Security Scanner* or *ITS4* for short [41]. The tool does a lexical analysis building a token stream of the

code. Then the tokens are matched with known vulnerable functions in a database. The reason for not performing a deeper analysis with the help of syntactic analysis (parsing) is that such an analysis cannot be made on the fly during programming. ITS4 is built to give developers support while coding, highlighting potential security problems as they are written. Parsing also suffers from being build dependent, not always covering the whole source code because of pre-processor conditionals.

When writing their paper the vulnerability database contained 131 potential vulnerabilities including problems with *race conditions* (not included in this paper, for reference see article by Bishop and Dilger [85]) and buffer overflows. *Pseudo random functions* are also considered risky since they're often used wrongly in security-critical applications. An entry in the database consists of:

- A brief description of the problem
- A high-level description of how to code around the problem.
- A grading of the vulnerability on the scale `NO_RISK`, `LOW_RISK`, `MODERATE_RISK`, `RISKY`, `VERY_RISKY`, `MOST_RISKY`.
- An indication of what type of analysis to perform whenever the function is found.
- Whether or not the function can retrieve input from an external source such as a file or a network connection.

ITS4 has a modular design which allows for integration in various development environments by replacing its front-end or back-end. In fact that was one of the design goals for ITS4. For the moment it only supports integration with GNU Emacs.

The ITS4 security tool is available for download on the Internet.
`http://www.cigital.com/its4/`

7.4.4 Flawfinder and Rats

Two new security testing tools where released in May 2001—*Flawfinder* developed by David A. Wheeler [42] and *Rough Auditing Tool for Security* (RATS) developed by Secure Software Solutions [43]. They both scan

source code on the lexical level, searching for security bugs. Their solutions are very similar to ITS4. When it was noticed that the two teams where developing similar tools they decided on a common release date and on trying to combine the two tools into one in the future.

Just as ITS4 Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, and more. The tool produces a list of potential vulnerabilities sorted by risk. This risk level depends not only on the function, but on the values of the parameters of the function. For example, constant strings are considered less risky than fully variable strings. The Flawfinder 0.19 vulnerability database contains 55 C security bugs.

RATS scans not only C and C++ code but also Perl, PHP and Python source code and flags common security bugs such as buffer overflows and race conditions. Just as Flawfinder and ITS4, RATS has a database of vulnerabilities and sorts found security bugs by risk. The RATS 1.3 vulnerability database contains 102 C security bugs.

Both these security testing tools are invoked from a shell with source code as input. They traverse the code and produce output with risk grading and short descriptions of the potential problems.

The security tools Flawfinder and RATS are available for download on the Internet.

`http://www.dwheeler.com/flawfinder/`

`http://www.securesw.com/rats/`

7.4.5 Splint

The next static analysis tool we describe is *LCLint* implemented by David Evans et al [86, 87]. The name and some of its functionality originates from a popular static analysis tool for C called *Lint* released in the seventies [88]. LCLint has later been enhanced to search for security specific bugs [40] and the first of January 2002 LCLint got the name *Secure Programming Lint* or *Splint* for short.

The Splint approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis on the syntactic level, making use of the program's parse tree. This means that the tool has a

much better chance of differentiating between correct and incorrect use of functions than the tools working on the lexical level.

The annotations specify function constraints in the program—what a function *requires* and *ensures*. Here is a simplified example from the annotated library `standard.h` in the Splint package:

```
char *strcpy (char *s1, char *s2)
      /*@requires maxSet(s1) >= maxRead(s2) @*/
      /*@ensures  maxRead(s1) == maxRead (s2) @*/
```

The `requires` clause specifies that buffer `s1` must be big enough to hold all characters readable from buffer `s2`. The `ensures` clause says that, upon return, the length of buffer `s1` is equal to the length of buffer `s2`. If a program contains a call to `strcpy()` with a destination buffer `s1` smaller than the source buffer `s2`, a buffer overflow vulnerability is present and Splint should report the bug.

To detect bugs the constraints in the annotations have to be resolved. Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by cojoining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if the latter hold. If they do not Splint issues a warning.

Note that we will not add any annotations to our test source code since that would be a violation of the second testing constraint defined in section 7.4. We rely fully on Splint's annotated libraries to make a fair comparison.

The Splint security tool is available for download on the Internet.
`http://www.splint.org/`

7.4.6 BOON

David Wagner et al presented a tool in 2000 describing aiming for detecting buffer overflow vulnerabilities in C code [61]. In July 2002 their tool, or rather working prototype, was publicly released under the name *BOON* which stands for *Buffer Overrun detectiON*. Under the assumption that most buffer overflows are in string buffers they model string variables (i.e.

the string buffers) as two properties—the allocated size, and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties of the string variable. The constraints are solved and matched to detect inconsistencies similarly to Splint.

Before analyzing the source code you have to use the C preprocessor on it to expand all macros and #include's. Then BOON parses the code and reports any detected vulnerabilities as belonging to one of three categories, namely "Almost certainly a buffer overflow", "Possibly a buffer overflow" and "Slight chance of a buffer overflow". The user needs to go check the source code by hand and see whether it is a real buffer overflow or not. Note that BOON does not detect format string vulnerabilities and is thus not tested for that.

The BOON security tool is available for download on the Internet.

`http://www.cs.berkeley.edu/~daw/boon/`

7.4.7 Other Static Solutions

There are several other approaches to static intrusion prevention. The area connects to general software testing which provides a broad range of potential methodologies.

A tool yet to be published is *Czech* by Jose Nazario [89]. Czech is a C source code checking tool that will do full out static analysis and variable tainting.

Software Fault Injection

A technique originally used in hardware testing called *fault injection* has also been used to find errors in software [45]. This has been used for security testing. By injecting faults, the system being tested is forced into an anomalous state during execution and the effects on system security is observed and evaluated.

Anup Ghosh et al implemented a prototype tool called *Fault Injection Security Tool*, or FIST for short [46]. The tool shows promising results but preparations of the source code have to be made by hand which means that the process is not automated. Also FIST is not available for download so we have excluded it from our analysis.

Also Wenliang Du and Aditya Mathur have done research on software fault injection for security testing [47]. They inject faults from the environment of the application, i.e. anomalous user input, erroneous environment variables and so on. In their paper they describe a methodology not yet implemented. Therefore their approach is not part of our analysis.

Constraint-Based Testing

Umesh Shankar et al from University of California at Berkeley present an interesting solution to finding format string vulnerabilities [90]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source. Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned of the possible vulnerability. Sadly, we did not manage to get their tool to report any vulnerabilities with the supplied annotated library functions.

7.5 Comparison of Static Intrusion Prevention Tools

Our testbed contains 20 vulnerable functions chosen from ITS4's vulnerability database (category `RISKY` to `MOST_RISKY`), Secure programming for Linux and UNIX HOWTO [80], and the whole `[fvsn]printf()`-family (see section 7.3.3 and 7.3.5 for a complete list). We do not claim that this test suite is perfectly fair, nor complete. But the sources from where we have chosen the vulnerabilities seem reasonable and the test result will at least provide us with an interesting comparison. Our 20 vulnerable functions are used in 13 safe buffer writings, 15 unsafe buffer writings, 8 safe format string calls and 8 unsafe format string calls, in total 44 function calls. We did not go into complex constructs to implement the safe function calls, rather a straight forward solution. An example of the difference between safe and unsafe calls is shown below:

```
char buffer[BUFSIZE];
```

	Flawf.	ITS4	RATS	Splint	BOON *
True Positives	22 (96%)	21 (91%)	19 (83%)	7 (30%)	4 (27%)
False Positives	15 (71%)	11 (52%)	14 (67%)	4 (19%)	4 (31%)
True Negatives	6 (29%)	10 (48%)	7 (33%)	17 (81%)	9 (69%)
False Negatives	1 (4%)	2 (9%)	4 (17%)	16 (70%)	11 (73%)

Table 7.1: Overall effectiveness and accuracy of static intrusion prevention. "Positive" means a warning was issued, "Negative" means no warning was issued. In total 44 function calls, 23 unsafe and 21 safe. * BOON only tested with buffer overflow vulnerabilities.

```
if(strlen(input_string)<BUFSIZE)
  strcpy(buffer, input_string); /* Safe */
strcpy(buffer, input_string);   /* Unsafe */
```

Overall results from our tests is presented in table 7.1 and detailed results are presented in table 7.2. The source code in short form can be found in Appendix .3. The exact source code and the print-outs from the various testing tools can be found on our homepage:
`http://www.ida.liu.se/~johwi`

7.5.1 Observations and Conclusions

As you would think all three lexical testing tools ITS4, Flawfinder and RATS, perform about the same on the true positive side. After all, a great part of our tested vulnerabilities where found in their databases or in publications connected to them, as stated before. But they differ considerably on the false positives where ITS4 is best.

For security aware programmers with knowledge of how buffer overflow and format string attacks work these tools can be very helpful. They will most probably get minor testing output, be able to sort out what is important and most importantly know how to solve the reported problems. For less experienced programmers the output might be too large and since the tools give no instructions on how to solve the problems they will need some other form of help.

Quite interesting is that Splint and BOON finds so few bugs. We contacted Splint author David Larochelle concerning this and he responded that the undetected bugs where not considered a serious threat since they are known to the security community and easily found with the UNIX command `grep`. We disagree with him—why not detect as many security bugs as possible? And why not help the developers that are not aware of the security vulnerabilities coming from misuse of several C functions?

Splint is the only tool that can distinguish between safe and unsafe calls to `strcat()` and `strcpy()`. This implicates that Splint has a good possibility to accurately detect security bugs with a low rate of false positives, just as you would think considering its deeper analysis of the code.

The general feeling we get after running the constraint-based testing tools is that they are still in some kind of a prototype state. Splint has been around under the name LCLint for some time and is used for general syntactical and semantical testing. But the security part needs to be completed. BOON is published as a prototype and should of course be judged as such.

None of the tools has high enough true positives combined with low enough false positives. Our conclusion is that none of them can really give the programmer peace of mind. And combining their output would be tedious.

7.6 Related Work

We have found one comparative study made of static intrusion prevention tools—"Source Code Scanners for Better Code" [91] by Jose Nazario. He compares the result from ITS4, Flawfinder and RATS when testing a part of the source code for OpenLDAP known to be vulnerable. It only contains one call to one of our 23 vulnerable functions—`vsprintf()`. No test for false positives is done either.

A study with another focus but relating to ours is "A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services" by Broadwell and Ong [92]. They investigate the strengths of static analysis versus software fault injection in finding errors in several large software packages such as Apache and MySQL. In static analysis they

Vulnerable Function	Flawf.		ITS4		RATS		Splint		BOON	
	T	F	T	F	T	F	T	F	T	F
gets()	1	-	1	-	1	-	1	-	1	-
scanf()	1	0	1	0	1	1	0	0	0	0
fscanf()	1	0	1	0	1	1	0	0	0	0
sscanf()	1	0	1	0	1	1	0	0	0	0
vscanf()	1	0	1	0	1	1	0	0	0	0
vsscanf()	1	0	1	0	1	1	0	0	0	0
vfscanf()	1	0	1	0	1	1	0	0	0	0
cuserid()	0	-	1	-	1	-	0	-	0	-
sprintf()	1	1	1	0	1	1	0	0	1	1
strcat()	1	1	1	1	1	1	1	0	1	1
strcpy()	1	1	1	1	1	1	1	0	1	1
streadd()	1	1	1	1	1	0	0	0	0	0
strecpy()	1	1	1	1	1	0	0	0	0	0
vsprintf()	1	1	1	0	1	1	1	1	0	0
strtrns()	1	1	1	1	1	0	0	0	0	0
printf()	1	1	1	1	1	1	1	1	-	-
fprintf()	1	1	1	1	1	1	1	1	-	-
sprintf()	1	1	1	1	1	1	1	1	-	-
snprintf()	1	1	1	1	0	0	0	0	-	-
vprintf()	1	1	0	0	0	0	0	0	-	-
vfprintf()	1	1	0	0	0	0	0	0	-	-
vsprintf()	1	1	1	1	1	1	0	0	-	-
vsnprintf()	1	1	1	1	0	0	0	0	-	-

Table 7.2: Detailed effectiveness and accuracy of intrusion prevention. T = 1 means an unsafe call was found (a true positive), F = 1 means a safe function call was deemed unsafe (a false positive). “-” means no such test is possible.

use ITS4 to find race conditions and BOON to find buffer overflows.

7.7 Conclusions

We have shown that the current state of static intrusion prevention tools is not satisfying. Tools building on lexical analysis produce too many false positives leading to manual work, and tools building on deeper analysis on syntactical and semantical level produce too many false negatives leading to security risks. Thus the main usage for these tools would be as support during development and code auditing, not as a substitute for manual debugging and testing.

Chapter 8

Modeling and Visualizing Security Properties of Code using Dependence Graphs[1]

8.1 Abstract

In this paper we discuss the problem of modeling security properties, including what we call the *dual modeling problem*, and ranking of potential vulnerabilities. The discussion is based on the results of a brief survey of eight existing static analysis tools and our own experience. We propose dependence graphs decorated with type and range information as a generic way of modeling security properties of code. These models can be used to characterize both good and bad programming practice as shown by our examples. They can also be used to visually explain code properties to the programmer. Finally, they can be used for pattern matching in static

[1]To be published in the Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden, 2005. Authors: John Wilander [4].

security analysis of code.

Keywords: security properties; dependence graphs; static analysis

8.2 Introduction

According to statistics from CERT Coordination Center, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software [7]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [8]. One way of countermeasuring these problems is using security tools to find the vulnerabilities already during software development.

In recent years a lot of research has been done in the field of static analysis for security testing. This research has resulted in several tools and prototypes based on various techniques, models and user involvement. Some of them are publicly available, some are not.

In November 2002 we published a comparative study of five tools publicly available at the time [3]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed. Although the research behind these tools and prototypes is often excellent and the empirical results are promising, it is not evident if and how the techniques can be combined to solve several security problems at once. They all focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user or programmer involvement. Further, to our knowledge there is no thorough study of the problems in modeling security properties that underlie static analysis.

8.2.1 Paper Overview

In Section 8.3 we present related work by doing a brief survey of eight existing static analysis tools performing syntactical and semantical static analysis to check security properties. A summary defines the problems we want to solve.

Graph models of security properties in code as a mean for visual communication with programmers is discussed in Section 8.4. Section 8.5 provides a definition and discussion of the *dual modeling problem* in the context of security properties in code. Criteria for severity ranking of security vulnerabilities are listed in Section 8.6.

In Section 8.7 we propose a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are explained in Section 8.8 and serve as examples of how the modeling formalism can be used.

Sections 8.9 and 8.10 discuss future work and provide our conclusions.

8.3 Survey of Static Analysis Tools

Static analysis tools try to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them. The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem. This paper tries to address these two drawbacks by proposing a way to model security properties of code that allows for both effective static analysis and visual communication with the programmer.

Several tools perform a deep analysis on a syntactical and semantical level. We have found eight such tools, all analyzing C code—Splint, BOON, CQual, Metal/xgcc, MOPS, IPSSA, Mjolnir, and Eau Claire. As some of these tools are still being developed and some are not even available as prototypes we do not know to what extent they are used in practice.

8.3.1 Splint

Secure Programming Lint or *Splint* was implemented by David Larochelle and David Evans [40].

Their approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis, making use of the program's parse tree. The annotations specify function constraints in the program—what a function *requires* and *ensures*.

Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by co-joining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if the latter hold. Splint only performs intraprocedural data-flow analysis, and the control-flow analysis is limited.

8.3.2 BOON

David Wagner et al presented *Buffer Overrun detectiON*, or *BOON*, aiming for detection of buffer overflow vulnerabilities [61]. In July 2002 a prototype was publicly released under the name . Under the assumption that most buffer overflows are in string buffers they model string variables (i.e. the string buffers) as abstract data types consisting of the allocated size and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties. Analysis is carried out by solving integer range constraints.

BOON reports any detected vulnerabilities as belonging to one of three categories, namely "Almost certainly a buffer overflow", "Possibly a buffer overflow" and "Slight chance of a buffer overflow".

8.3.3 Cqual

The tool *Cqual* uses constraint-based type inference [93]. It traverses the program's abstract syntax tree and generates constraints that capture the relations between type qualifiers. A solution to the constraints gives a

Table 8.1: Overview of static analysis tools checking C code for various security properties (cont.). "Intra" and "Inter" = intra- or interprocedural analysis, "Alias" = data aliasing, "Ptr" = pointer analysis, "Type" = type and type conversion information, and "Annot" = code annotations.

	Control-flow		Data-flow					Annot
Tool	Intra	Inter	Intra	Inter	Alias	Ptr	Type	
Splint	x		x		x			x
BOON			x	x				
Cqual			x		x		x	x
MOPS	x	x						
Metal/xgcc	x	x	x	x				
IPSSA	x	x	x	x	x	x	x	
Mjolnir	x	x	x	x	x			
Eau Claire	x	x	x	x				

valid assignment of type qualifiers to the variables in the program. If the constraints have no solution, then there is a potential bug.

Umesh Shankar et al have used Cqual to find format string vulnerabilities [90]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source (Cqual requires the user to manually tag untrustworthy data sources). Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned.

The same tainted functionality was used by Chen et al to statically find implicit type cast errors constituting security vulnerabilities [94]. Johnson and Wagner are using Cqual to check for insecure pointer handling between kernel and user-space in Linux [95].

8.3.4 Metal and xgcc

Ashcraft and Engler have done security research in the area of meta-level compilation. With their compiler extension *xgcc* and extension language

Metal they have statically analyzed code for input validation errors on integer variables [96]. C programs are modeled as control-flow graphs and are analyzed path by path.

By formulating rules in Metal they check that integer values coming from untrusted sources are bounds checked before they are used in any sensitive function. The security bugs found are unvalidated integers used in pointer arithmetic, and integer overflows. Memory management errors (`malloc()/free()`) were also found but not substantially analyzed.

Potential bugs found are ranked by properties such as local vs global scope, distance in lines of code, and non-aliased vs aliased variables.

8.3.5 MOPS

Chen and Wagner have designed a static analysis tool called MOPS which checks ordering constraints [97]. Some security bugs can be described in terms of temporal safety properties. MOPS specifically checks dropping of privileges and race conditions in file accesses. C programs are modeled as *push-down automata*, and the security properties are modeled as *finite state automata*. Security models can be combined into complex security properties.

No data-flow, pointer, or aliasing analysis is done, which is justifiable since only temporal properties are checked.

8.3.6 IPSSA

Livshits and Lam have defined and used an extended intermediate form for finding buffer overflow and format string bugs [98]. Their program model builds on *static single assignment* (*SSA*) form—an intermediate code representation that separates values operated on from the locations they are stored in which is very useful in for instance optimization [99]. The extension, called *IPSSA*, provides interprocedural definition-use information with indirect memory accesses via pointers. It can then be used to perform static analysis that handles pointer and aliasing analysis. Security properties are modeled using a "small special-purpose language designed for the purpose". While technical details of this special-purpose language are lacking their empirical results are very promising, especially the low rate

of false positives. Their solution was chosen to be unsound for scalability reasons.

8.3.7 Mjolnir

Weber et al have presented a tool called *Mjolnir* which makes use of dependence graphs and constraint solving to find buffer overflows in C code [100]. They represent buffers with the same range variables used in BOON (see Section 8.3.2), build system dependence graphs, decorate them with range constraints based on the semantics of C string library functions, and finally solve the constraint sets.

To decorate the dependence graphs they traverse the program bottom-up and generate summary nodes containing the constraints of the current function and all its callees.

Weber et al do not clearly state how safety constraints are generated, but we assume they generate them only for statically allocated buffers. They provide both control-flow insensitive and control-flow sensitive constraint generation. Although global variables normally are handled in dependence graphs (see Section 8.7.1) they are not handled by Mjolnir. No pointer analysis is done.

8.3.8 Eau Claire

In spring 2002 Brian Chess presented his tool *Eau Claire* [101]. The tool translates C code into so called *guarded commands*, enhanced with exceptions, assertions, assume statements, and erroneous states. Vulnerabilities are modeled using the ESC/Modula2 specification language where you define what a function requires, modifies, and ensures. Eau Claire then augments guarded commands with the specifications. The outcome is a set of verification conditions which are processed by an automatic theorem prover to find potential violations.

Shortcomings of Eau Claire's static analysis are the conservative approach to pointer dereferences (it assumes that any two pointers of the same type may reference the same location) and references into structures and unions. Type-based vulnerabilities are not targeted by Eau Claire [102].

Table 8.2: Overview of static analysis tools checking C code for various security properties. The program models are control-flow graph (CFG), abstract syntax tree (AST), push-down automata (PDA), parse tree (PST), static single assignment (SSA), system dependence graph (SDG), guarded commands (GC).

	Program model						
Tool	CFG	AST	PDA	PST	SSA	SDG	GC
Splint	x						
BOON				x			
Cqual		x					
MOPS			x				
Metal/xgcc	x						
IPSSA					x		
Mjolnir						x	
Eau Claire							x

Table 8.3: (Continued) Overview of static analysis tools checking C code for various security properties. The property models are constraint based (CB), finite state automata (FSA), "Metal" (MET), ESC/Modula2 specification language (ESC), and other, special purpose modeling (OTH).

	Security property model				
Tool	CSB	FSA	MET	ESC	OTH
Splint	x				
BOON	x				
Cqual	x				
MOPS		x			
Metal/xgcc			x		
IPSSA					x
Mjolnir	x				
Eau Claire				x	

8.3.9 Summary

Tables 8.1, 8.2, and 8.3 summarize the properties and features of the tools above.

We conclude that several categories of security properties can be statically checked but there is need of a generic solution. The first step toward such a solution is to define a modeling formalism that both covers all necessary aspects and allows for static analysis.

Two other key issues are that such a solution has to allow for effective feedback to the programmers who have to fix the security problems, and it has to support intuitive modeling of new security properties for effective updates of the database. None of the tools presented above have any other kind of input or feedback than text.

We require that the modeling formalism can:

- visually communicate with programmers who model or fix security problems in code (Section 8.4);
- model several types of security properties (Section 8.5);
- rank the severity of potential flaws (Section 8.6); and
- take into account data-flow, control-flow, type and range information, and combinations thereof (Section 8.7).

8.4 The Need for Visual Models

As mentioned in Section 8.3 the two main drawbacks of static analysis tools are that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem.

Current tools such as the ones briefly presented in Section 8.3 use textual models of security properties in their databases to give textual feedback to the user. For example Splint gives output in the following manner:

```
bounds.c:9: Possible out-of-bounds store:
  strcpy(str, tmp)
  Unable to resolve constraint:
```

```
requires maxSet(str @ bounds.c:9) >=
maxRead(getenv("MYENV") @ bounds.c:7)
 needed to satisfy precondition:
requires maxSet(str @ bounds.c:9) >=
maxRead(tmp @ bounds.c:9)
 derived from strcpy precondition: requires
maxSet(<parameter 1>) >=
maxRead(<parameter 2>)
```

Just as call graphs and and flow graphs can help programmers understand code in general (Grammatech's tool "CodeSurfer" is a perfect example [103]), visual models and graph representations of security properties can help to understand and fix security flaws. Especially when the flaws include interprocedural data- and control-flow dependencies.

8.5 The Dual Modeling Problem

A common issue in security modeling is what we call the *dual modeling problem*—the problem of modeling malicious or benign things. When modeling security properties of code we need both kinds—models of bad programming practice, and models of good programming practice.

In a seminal paper from 1977 Leslie Lamport describes a formalism closely related to the dual modeling problem—a property stating that nothing bad happens during execution is called a *safety property*, and a property stating that something good (eventually) happens during execution is called a *liveness property* [104].

Typical for a safety property is that we can detect a property violation between one execution step and another. During execution we can look ahead and see if the next execution step will take us into a bad state and in such a case raise an alarm or terminate execution. All run-time security measures such as intrusion detection systems and anti-virus applications detect safety properties—they either try to match with known bad behavior, or they monitor for deviations from good behavior.

In the case of a liveness property we can only detect property violations at termination since during execution, we never know whether the good thing will eventually happen or not. Fulfilling the liveness property could potentially be the last execution step before termination. Therefore

we cannot rely on run-time monitoring to countermeasure security vulnerabilities that are violations of liveness properties. Static methods, on the contrary, can look into the "future" by following possible execution paths all the way to termination, and try check if a program satisfies a liveness property.

However, models of good or bad programming practice do not correspond directly to safety and liveness properties. Instead they can be a combination of safety and liveness as explained in Section 8.5.1 and 8.5.2.

A comprehensive discussion on this fundamental difference between safety and liveness security properties can be found in Schneider's paper "Enforceable Security Policies" [105].

8.5.1 Modeling Good Security Properties

Some security properties of code are typically described as "If you do A you must do B". These properties are best modeled as good programming practice—"do like this".

An example is *input validation* of integers. When an integer can be affected by input from users, files, the network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

While being a model of good programming practice correct input validation is both a liveness property (external input must eventually be validated assuming it will be used sometime), and a safety property (no sensitive use of external input without validation).

8.5.2 Modeling Bad Security Properties

Some security problems are typically described as "If you do A then you must not do B". Such properties are best modeled as bad programming practice—"do not do like this".

An example of such a problem is the double `free()` vulnerability. Freeing the same memory chunk twice or more may open up for heap corruption attacks.

Trying to model all possible benign ways of freeing memory is infeasible since that would be the same as building complete models of all well-behaved programs using `free()`. A model of a bug, however, covers all cases. The absence of multiple `free()` is a safety property.

8.6 Ranking of Potential Vulnerabilities

Engler and Musuvathi have clearly pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—"It's not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ..." [106]. Based on our knowledge and experience on static analysis we propose using the following information from the analysis to generate severity ranking:

- Pointer analysis is a hard problem to solve accurately and thus the risk for false positives increases with the amount of such analysis. Therefore we propose that the more pointer analysis involved in finding a flaw, the lower the ranking.
- Aliasing is another problem in static analysis. Because of potential inaccuracy in the analysis we therefore propose that the more aliasing involved in finding a flaw, the lower the ranking.
- Interprocedural control-flow may result in infeasible execution paths being analyzed. Again, because of potential inaccuracy in the analysis, flaws involving interprocedural analysis are ranked lower than intraprocedural ones.
- Flaws involving implicit events are ranked higher than explicit ones since implicity imposes a higher risk for unintended behavior. An example of this is implicit versus explicit type-casts.

8.6.1 Using the Dual Model for Ranking

In some cases we can make use of modeling both good and bad programming practice. If we have reached a concise description of a property in one distinct model, the dual of that model often explodes into several cases.

For instance, in the case of implicit type-casting and integer signedness vulnerabilities a model of good programming practice is to validate the integer and to have no implicit type-casts at any use points (this example is explained in detailed in Section 8.8.1).

The dual of this model contains several ways of violating the property. Various narrowing type-casts and missing validation points can be combined. The benefit of exploding the dual and creating all these models is that we can possibly rank them in terms of severity. Perhaps a certain violation is definitely a security vulnerability, whereas another violation only *might* be vulnerable.

8.7 A More Generic Modeling Formalism

To meet the requirements listed in Section 8.3.9 we propose decorated dependence graphs as a more generic formalism for visualizing and modeling security properties, and performing static analysis. We here present intraprocedural and interprocedural dependence graphs, decorated with range and type information. We end the section with a view on possible analysis techniques.

8.7.1 Program Dependence Graphs

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [107]. While originally generated for procedural languages such as C, algorithms generating dependence graphs for object oriented languages exist, e.g. Java [108]

A dependence graph is an intermediate representation of code where vertices represent statements and predicates (henceforth called *program points*), and edges represent control- and data-flow *dependence*. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph.

A program point B is *control dependent* on another program point A, if A controls whether B is executed or not. Formally A is the first program point not *post-dominated* by B when traversing the control-flow graph back-

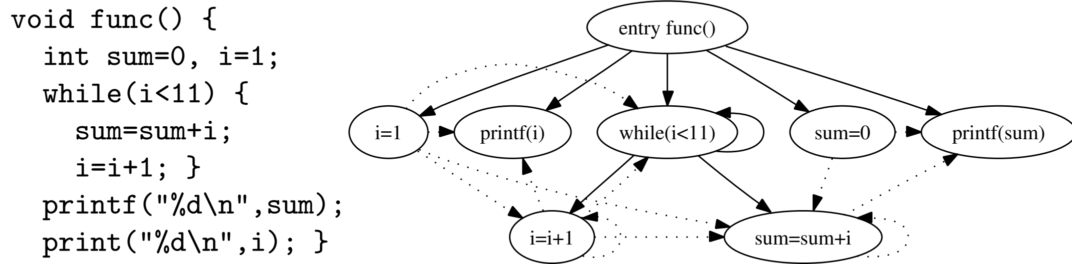


Figure 8.1: A small C function (left) with its corresponding program dependence graph (right). Solid arrows represent control-flow dependence, dotted arrows represent data-flow dependence. All dependencies are transitive (if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$).

ward from B. Informally we can say that program point A is a conditional and B is executed in only one of A's outgoing paths.

A program point B is *data dependent* on a program point A if some variable x is defined in A and later used in B without any new defines in-between. Data dependence can also be in form of definition order. Figure 8.1 shows a small C function with its corresponding program dependence graph.

8.7.2 System Dependence Graphs

The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [109]. To generate the SDG we need to encode data- and control-flow dependence between procedures which includes formal and actual parameters, formal and actual return values, and global variables.

A procedure call from procedure A to procedure B is modeled with a call vertex in A, an entry vertex in B, and an interprocedural control dependence edge between them. Parameters are handled with actual-in and actual-out vertices in A, formal-in and formal-out vertices in B, and interprocedural data dependence edges connecting them. Temporary variables are used for parameter passing by value-result. If a procedure uses a global variable, it is treated as a (hidden) input parameter, and is encoded

as additional actual-in and formal-in vertices. For further information on summary edges for avoiding calling context problems see the original paper [109].

8.7.3 Range Constraints in SDGs

Weber *et al* have used decorated SDGs to statically detect buffer overflow vulnerabilities [100]. The graph is augmented with range constraint information for string buffers. Each PDG contains a summary vertex with range constraints of the procedure and all its callees.

```
void copy(char *src) {
  char dst[10];
  strcpy(dst, src); }
```

The PDG for the code to the left would have a range constraint node summary node saying Len(`src`) $\subseteq$ Len(`dst`).

8.7.4 Type Information in SDGs

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [94].

We propose that the original SDGs be decorated with type information, specifically implicit type conversions. Type conversion information should belong to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others. See Figure 8.5 and 8.5 for examples of this decoration.

8.7.5 Static Analysis Using SDGs

Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* and *chopping* and are used for optimization [110].

A program slice is the parts of a program that can affect the value of a chosen program point, the *slicing criterion*. *Static slicing*, invented by Weiser [111], was defined as a reachability problem in PDGs by Ottenstein

and Ottenstein [107]. Interprocedural slices can be computed in a similar way in SDGs.

The combination of two (or more) program points, potentially a point with (malicious) user input, and a point with a vulnerability, allows for program chopping—a technique presented by Reps *et al* [112]. When chopping we want to know how some source points affect some target points.

Slices and chops of programs can help with understanding the cause of a vulnerability since they show exactly what parts of the program affect the execution of the vulnerable program point. The richness of program information found in SDGs together with slicing, chopping, type inference and range analysis means it covers all the features of the tools surveyed in Section 8.3 and provides visual communication with the user via a graph representation of the original code.

8.8 Modeling Security Properties

In this section we show how four security properties can be modeled in terms of decorated dependence graphs. We show the use of dual models both for benign and malicious properties, and ranking of potential flaws. Our proposed formalism is not limited to these properties; they simply serve as examples.

In the graphs all edges represent interprocedural transitive dependence—solid arrows for control-flow, and dotted arrows for data-flow.

8.8.1 Integer Flaws

Handling integers may seem harmless and straight forward. But several security vulnerabilities prove this a difficult area. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and when the integer representation changes from signed to unsigned or vice versa. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing typecasts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

```
void func1(char *dest, char *src,
           int len) {
  if(len<MAX)
    memcpy(dest, scr, len); }
```

Figure 8.2: Implicit type-cast flaw (`len` casted to unsigned int in the call to `memcpy()`).

```
void func2(unsigned int size) {
  char *buf =
       (char *) malloc(size+1);
}
```

Figure 8.3: Integer overflow flaw (adding one to `size` may cause overflow).

We are now able to encode the first correct code pattern in terms of our decorated dependence graphs (see Fig. 8.4). The nodes are program points where “ext input” means external input, “def” means a variable is defined, “val” means a variable is validated, and “use” means a variable is used. The input has to be validated before it is used which means that the use point has to be control dependent on the validation point.

Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [96].

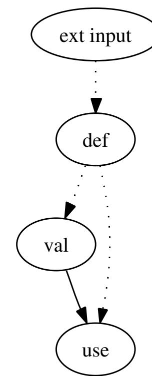


Figure 8.4: Correct code pattern for integer input validation.

Deviations from this good programming practice, i.e. integer security bugs, have been studied by Blexim [113], Howard [114], and Ashcraft and Engler [96] and we here briefly present the bug types they have identified:

Integer Signedness Errors.

Integer signedness errors can arise both due to implicit type-casting and insufficient validation. In Fig. 8.2 the signed integer `len` can be negative and

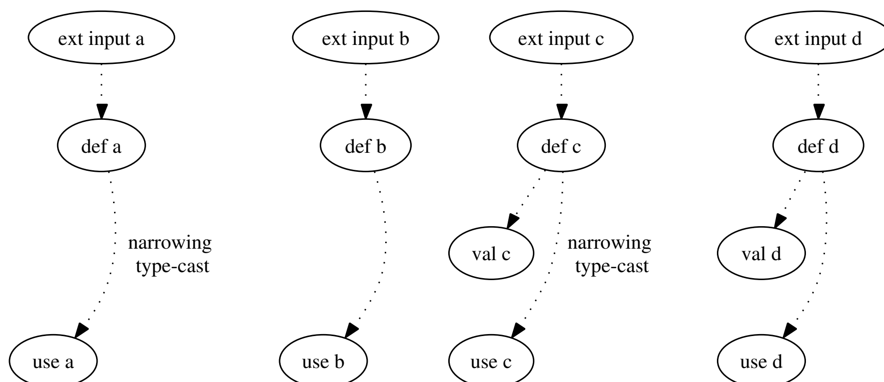


Figure 8.5: Four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). The proposed severity ranking from left to right is explained in Section 8.8.2.

as such pass the (inadequate) validation point. When calling `memcpy()` an implicit narrowing type-cast to `size_t` (unsigned integer) occurs which will convert a negative integer to a huge positive integer, possibly overflowing the destination buffer `dest`.

Integer Overflow/Underflow.

When an unsigned integer has reached the maximum value it can represent, an increment to that integer will make it wrap around and become zero. Decrementing an unsigned integer below zero will result in the maximum value.

In Fig. 8.3 the intent is to allocate the requested memory plus space for a null terminator. If `size` was the maximum unsigned integer possible, adding one will make it wrap around and call `malloc()` with zero as argument. The return value in such a case is either a null pointer or a non-null pointer that must not be used. Dereferencing such a non-null pointer may allow for heap corruption.

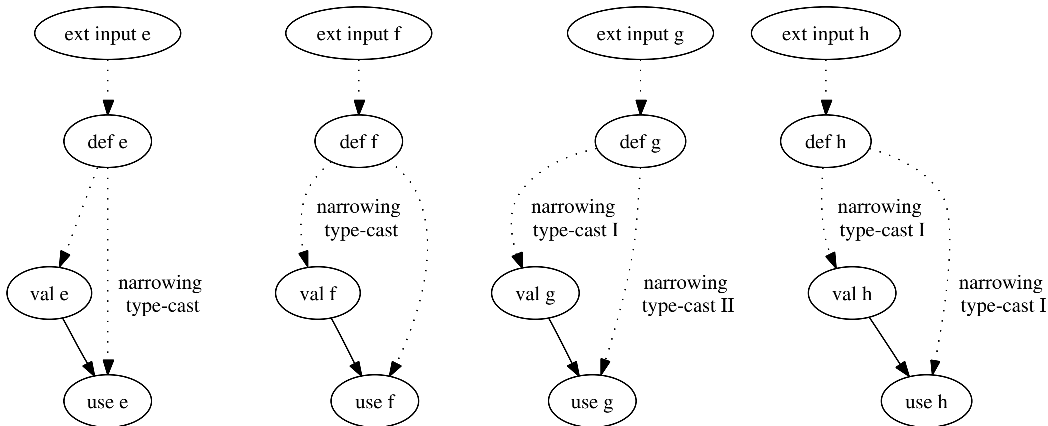


Figure 8.6: (Cont.) Four out of eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). "narrowing type-cast I" and "narrowing type-cast II" means two *different* type-casts. The proposed severity ranking from left to right is explained in Section 8.8.2.

Integer Input Validation.

When an integer can be affected by input from users, files, network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

8.8.2 Modeling Integer Flaws

To allow for severity ranking we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 8.5 and 8.6). Using the ranking rule for implicity (see Section 8.6) we rank the incorrect code patterns in descending order as follows:

1. Missing validation and narrowing type-cast
2. Missing validation but no narrowing type-cast
3. Use not control dependent on validation and narrowing type-cast
4. Use not control dependent on validation but no narrowing type-cast
5. Narrowing type-cast on either validation or use (two graphs in Fig. 8.6)
6. Different narrowing type-casts on validation and use
7. Same narrowing type-casts on validation and use

8.8.3 The Double `free()` Flaw

Often "normal" bugs turn out to be tools for attackers. This is the case of *double free*. To allocate heap memory, the program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`.

To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored "in-band". When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault*. But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of a model of a bad security property (see Fig. 8.7). We show in Fig. 8.8 and 8.9 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`; we also have to look for bad use of `free()`.

```
char *buf = (char *) malloc(SIZE);
...
free(buf);
...
free(buf);
```

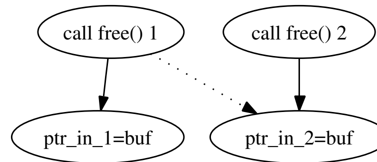


Figure 8.7: Incorrect code pattern for `free()` and the corresponding dependence graph. If there had been a new call to `malloc()` in-between the two calls to `free()` there would not have been a data dependency edge between the first call to `free()` and the second pointer to `buf` in the graph.

8.8.4 Modeling External Input

Knowing which data sources not to trust is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. *Environment variables* are considered untrustworthy sources [115], and Ashcraft and Engler add another three categories—System calls, routines that copy data from user space, and network data [96]. In modeling security properties these sources of so called *tainted* data will all be considered as nodes of external input and analyzed via transitive data dependencies.

8.9 Future Work

Finding the modeling formalism is the first step toward a single tool able to check for several security properties. We are right now implementing a prototype tool called *GraphMatch* that uses dependence graphs to check security properties [116]. The prototype currently finds interprocedural input validation flaws. Apart from modeling other security properties and checking them with real-life code, we plan to investigate scalability and accuracy issues of the analysis, and also evaluate dependency graphs as a visual aid in secure programming. Empirical studies will be made to evaluate the heuristic ranking of potential vulnerabilities.

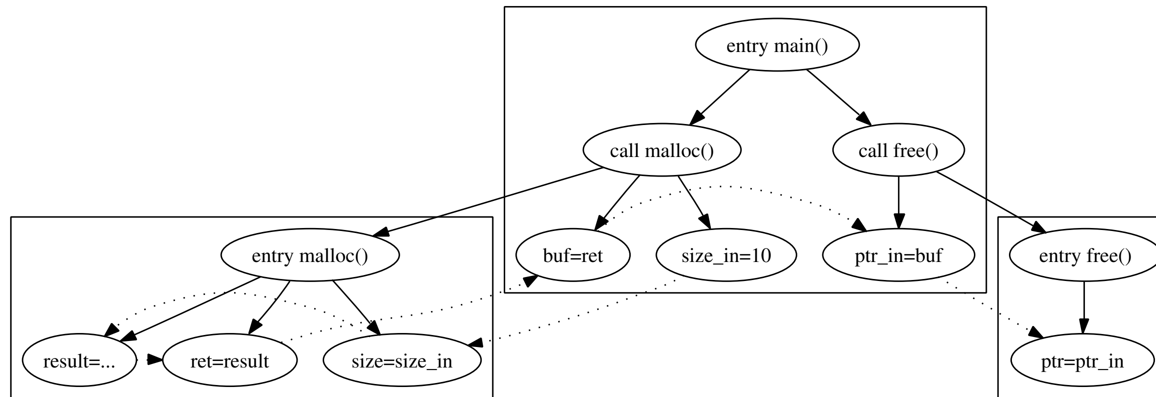


Figure 8.8: Correct graph pattern for `malloc()` and `free()`.

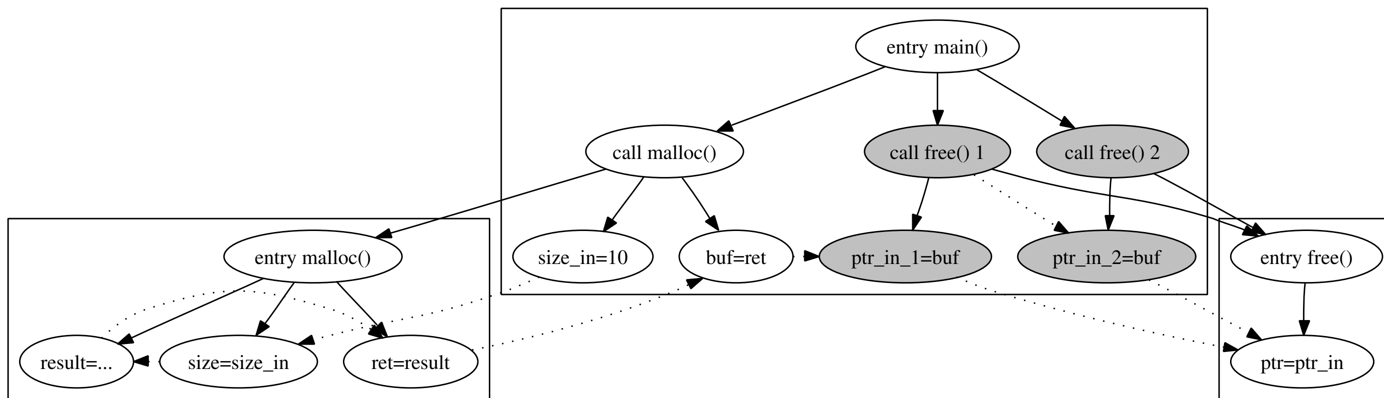


Figure 8.9: Incorrect graph pattern for `malloc()` and `free()`, where `free()` is called twice. Notice how the grey nodes in the `main()` box match the incorrect code pattern for `free()` which was shown in Fig. 8.7.

8.10 Conclusions

We have shown that there is a need for a generic formalism both for description of security properties and for static checking of these properties. In addition we believe that visual support is needed to effectively communicate with programmers. System dependence graphs decorated with range constraints and type conversion information can serve that purpose. Dependence graphs are well-known in the static analysis and compiler communities and are able to model the diversity of security properties, covering both safety and liveness properties of code, as shown by our examples.

8.11 Acknowledgments

We would like to sincerely thank the previewers of this paper, especially David Byers.

Chapter 9

Future Work

There are three general directions for future work—security requirements, run-time intrusion prevention, and compile-time intrusion prevention.

9.1 Security Requirements

Some new questions have come up during the analysis of the field study presented in Chapter 5. First of all a similar *case study*, where interaction with the organizations would be allowed, could answer the questions of risk analysis and prioritizing of requirements. How would a risk analysis affect the requirements? Such a study could also investigate the local heroes phenomenon by interviewing the people who formulate various security requirements.

Secondly, the systems in our field study could be evaluated in terms of security to see to what extent poor requirements are transformed into insecure systems. This naturally requires access to the systems.

Finally, a checklist or digital form could be developed to ensure more complete security requirements. Such a checklist could then be evaluated both by users and by studying the requirements specifications.

9.2 Run-Time Intrusion Prevention

We are currently on the way to develop a much larger testbed for evaluating run-time buffer overflow prevention techniques and tools. The original testbed comprised 20 attack forms built from three dimensions:

Techniques. Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations. The stack or the heap/BSS/data segment.

Attack Targets. The return address, the old base pointer, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Since then we have continuously collected information about new attack forms and have both enlarged the original three dimensions, and added four new dimensions. Our new testbed will be built from the following:

Techniques (same). Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations (same). The types of location for the buffer overflow are the stack or the heap/BSS/data segment.

Attack Targets (extended). The return address, the old base pointer, virtual pointers, exception handlers, malloc/free info, global offset table, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Overflow Function (new). 16 different library functions plus manual memory write via pointer and loop.

Vulnerable Buffer (new). `char` buffer or other.

Attack code (new). Injected code, “return into libc”, or injected parameters such as `/bin/sh` sent to `system()`.

Polymorphism (new). No `NOP` sled, normal `NOP` sled, or polymorphic `NOP` sled.

This aims at the order of 10.000 attack forms! Several new prevention techniques and tools have been developed and will be tested with our testbed (see Related Work, Chapter 4).

9.3 Compile-Time Intrusion Prevention

We are currently working on a prototype tool that uses dependency graphs to model security properties of C code and then perform pattern matching to find security bugs. The work so far is presented in Pia Fåk's Master's thesis [116]. A number of research questions are still open and will be the main focus in our further studies:

Complexity. Not too surprisingly, our initial results show that our graph matching has high complexity. It might be that dependency graph matching can be reduced to the *subgraph isomorphism problem* which is shown to be NP-complete. Even so, we will investigate how heuristic trade-offs leading to unsoundness and/or incompleteness can affect practical performance.

Accuracy. How much does the inevitable inaccuracy of the underlying program analysis affect the accuracy of our pattern matching?

Generality. Are dependency graphs suitable for modeling a great variety of security properties of code? Are they suitable for analysis of other languages than procedural ones such as C?

Usability. Can visualization of code properties with dependence graphs help the programmers fix vulnerable code? Can it help in secure programming education?

Heuristic Ranking. Can we find effective heuristics for ranking potential security bugs found?

Model Updates. Will our security property database be fairly static or will it need continuous updates with new flavors of the security properties?

Chapter 10

Summary and Conclusions

In this licentiate thesis we have focused on policy assurance and implementation assurance for software security.

To build more secure software, accurate and consistent security requirements must be specified. By doing a field study of eleven requirement specifications on IT systems we have shown that current practice in security requirements is poor. The specifications were inconsistent in selection of requirements, inconsistent in level of detail, and contained almost no requirements on standard security solutions.

To build more secure software we specifically need assurance requirements on code. A way to achieve implementation assurance is to use effective methods and tools that solve or warn for known vulnerability types.

Our comparative study of publicly available tools for run-time prevention of buffer overflow attacks shows that more has to been done to countermeasure the threat of such attacks. The best tool was effective against only 50 % of the attacks and there were six attack forms which none of the tools could handle.

We have also investigated the effectiveness of five publicly available compile-time intrusion prevention tools. The test results show high rates of false positives for the tools building on lexical analysis and low rates of true positives for the tools building on syntactical and semantical analysis.

As a first step toward a more effective and generic compile-time solu-

tion we have proposed dependence graphs decorated with type and range information as a way of modeling and pattern matching security properties of code. Apart from allowing static code analysis, these models can be used to characterize both good and bad programming practice. They can also be used to visually explain code properties to the programmer.

Chapter 11

Appendices

.1 Empirical Test of Dynamic Buffer Overflow Prevention

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Halted	Halted	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 1: **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Abnormal	Prevented	Missed
Libsafe and Libverify	Halted	Missed	Halted

Table 2: **(Continued) Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 3: **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Abnormal	Missed

Table 4: **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Missed	Missed

Table 5: **(Continued) Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Abnormal	Missed
Stack Shield Global Ret Stack	Prevented	Abnormal	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 6: **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Attack Target Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed	Missed
Stack Shield Global & Range	Missed	Missed	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 7: **(Continued) Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

.2 Theoretical Test of Dynamic Buffer Overflow Prevention

Attack Target Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Halted	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Missed	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Halted	Halted	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 8: **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Missed	Halted	Missed
Libsafe and Libverify	Halted	Missed	Halted

Table 9: **(Continued) Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 10: **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Missed	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Halted	Halted	Missed

Table 11: **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Missed	Missed

Table 12: **(Continued) Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable
StackGuard Terminator Canary	Missed	Halted	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted
Stack Shield Range Ret Check	Halted	Halted	Halted
Stack Shield Global & Range	Prevented	Prevented	Halted
ProPolice	Missed	Halted	Missed
Libsafe and Libverify	Halted	Halted	Missed

Table 13: **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Attack Target Development Tool	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed	Missed
Stack Shield Global Ret Stack	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Missed
Stack Shield Global & Range	Halted	Missed	Missed
ProPolice	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed

Table 14: **(Continued) Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

.3 Static Testbed for Instrusion Prevention Tools

In this appendix we have included the 44 function calls used to compare publicly available tools for static intrusion prevention. To shorten it down we have only included the interesting parts.

```
#define BUFSIZE 9
static char static_global_buffer = 'A';
static char global_buffer[BUFSIZE];

/***** Buffer Overflow Vulnerabilities *****/

pointer = gets(buffer); /* Unsafe */

scanf("%8s", buffer_safe);  /* Safe */
scanf("%s", buffer_unsafe); /* Unsafe */

fscanf(fopen(file_name, "w"), "%8s", buffer_safe);  /* Safe */
fscanf(fopen(file_name, "w"), "%s", buffer_unsafe); /* Unsafe */

sscanf(input_string, "%8s", buffer_safe);  /* Safe */
sscanf(input_string, "%s", buffer_unsafe); /* Unsafe */

if(choice==0) vscanf("%8s", arglist); /* Safe */
else vscanf("%s", arglist);           /* Unsafe */

if(choice==0) vsscanf(input_string, "%8s", arglist); /* Safe */
else vsscanf(input_string, "%s", arglist);           /* Unsafe */

if(choice==0)
  vfscanf(fopen(file_name, "w"), "%8s", arglist); /* Safe */
else
  vfscanf(fopen(file_name, "w"), "%s", arglist);  /* Unsafe */

sprintf(buffer_safe, "%8s", input_string);  /* Safe */
sprintf(buffer_unsafe, "%s", input_string); /* Unsafe */

if(strlen(input_string)<BUFSIZE)
  strcat(buffer_safe, input_string);   /* Safe */
strcat(buffer_unsafe, input_string);   /* Unsafe */

if(strlen(input_string)<BUFSIZE)
```

```
  strcpy(buffer_safe, input_string);   /* Safe */
strcpy(buffer_unsafe, input_string);   /* Unsafe */

cuserid(buffer_unsafe); /* Unsafe */

if(choice==0) vsprintf (buffer_safe, "%8s", arglist); /* Safe */
else vsprintf (buffer_unsafe, "%s", arglist);         /* Unsafe */

res = streadd(buffer_safe, "a", "");            /* Safe */
res = streadd(buffer_unsafe, input_string, ""); /* Unsafe */

res = strecpy(buffer_safe, "a", "");            /* Safe */
res = strecpy(buffer_unsafe, input_string, ""); /* Unsafe */

res = strtrns("a", "a", "A", buffer_safe);            /* Safe */
res = strtrns(input_string, "a", "A", buffer_unsafe); /* Unsafe */

/***** Format String Vulnerabilities *****/

printf(&static_global_buffer); /* Safe */
printf(global_buffer);         /* Unsafe */

fprintf(stdout, &static_global_buffer); /* Safe */
fprintf(stdout, global_buffer);         /* Unsafe */

char local_buffer[BUFSIZE];
/* Safe */
sprintf(local_buffer, &static_global_buffer, input_string);
/* Unsafe */
sprintf(local_buffer, global_buffer, input_string);

char local_buffer[BUFSIZE];
/* Safe */
snprintf(local_buffer, BUFSIZE, &static_global_buffer, input_string);
/* Unsafe */
snprintf(local_buffer, BUFSIZE, global_buffer, input_string);

if(choice==0) vprintf(&static_global_buffer, arglist); /* Safe */
else vprintf(global_buffer, arglist);                  /* Unsafe */

if(choice==0) /* Safe */
  vfprintf(stdout, &static_global_buffer, arglist);
else /* Unsafe */
  vfprintf(stdout, global_buffer, arglist);
```

```
char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsprintf(local_buffer, &static_global_buffer, arglist);
else /* Unsafe */
  vsprintf(local_buffer, global_buffer, arglist);

char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsnprintf(local_buffer, BUFSIZE, &static_global_buffer, arglist);
else /* Unsafe */
  vsnprintf(local_buffer, BUFSIZE, global_buffer, arglist);
```

Bibliography

[1] John Wilander and Jens Gustavsson. Security requirements—a field study of current practice. In *E-Proceedings of the Symposium on Requirements Engineering for Information Security, in conjunction with the 13th IEEE International Requirements Engineering Conference (to appear in formal proceedings)*, Paris, France, http://www.sreis.org, August 2005.

[2] John Wilander and Mariam Kamkar. A comparative study of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network & Distributed System Security Symposium*, San Diego, California, February 2003.

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

[4] John Wilander. Modeling and visualizing security properties of code using dependence graphs. In *Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden (to appear)*, Vasteras, Sweden, http://www.idt.mdh.se/serps-05/, October 2005.

[5] John Wilander. Security intrusions and intrusion prevention. Master's thesis, Linkopings universitet, http://www.ida.liu.se/~johwi, April 2002.

[6] Herbert H. Thompson and James A. Whittaker. Rethinking software security. *Dr. Dobb's Journal*, 29(2):73–75, February 2004.

[7] CERT Coordination Center. CERT/CC statistics 1988-2004. http://www.cert.org/stats/cert_stats.html, January 2005.

[8] CSO magazine, U.S. Secret Service, and CERT Coordination Center. 2004 e-crime watch survey. http://www.csoonline.com/releases/052004129_release.html, May 2004.

[9] John Viega and Gary McGraw. *Building Secure Software : How to Avoid Security Problems the Right Way*. Addison–Wesley, 2001.

[10] Computer Science and National Research Council Telecommunications Board. Cybersecurity today and tomorrow: Pay now or pay later (prepublication). Technical report, National Academies, USA, http://www.nap.edu/books/0309083125/html/, January 2002.

[11] Lisa M. Bowman. Companies on the hook for security. http://news.com.com/2100-1023-821266.html, January 2002.

[12] BBC News. Software security law call. http://news.bbc.co.uk/hi/english/sci/tech/newsid_1762000/1762261.stm, January 2002.

[13] Matt Bishop. *Computer Security : Art and Science*. Addison–Wesley, 2003.

[14] Anup K. Ghosh, Chuck Howell, and James A. Whittaker. Building software securely from the ground up. *IEEE Software*, 19(1):14–16, February 2002.

[15] International Organization for Standardization. ISO/IEC 17799:2000 information technology – code of practice for information security management. http://www.iso.org/iso/en/prods-services/popstds/informationsecurity.html.

[16] Tzi cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

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

[18] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proceedings of The 17th Large Installation Systems Administration Conference*, San Diego, USA, October 2003.

[19] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. Technical report, Columbia University, Computer Science Department, November 2003.

[20] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[21] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging AADEBUG'97*, Linkoping, Sweden, May 1997.

[22] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with StackGuard. Linux Expo `http://www.cse.ogi.edu/~crispin/`, May 1999.

[23] 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, USA, February 2004.

[24] Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[25] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[26] Danny Nebenzahl and Avishai Wool. Install-time vaccination of windows executables to defend against stacksmashing attacks. In *Proceedings of The 19th IFIP International Information Security Conference*, Toulouse, France, August 2004.

[27] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium*, San Diego, USA, February 2005.

[28] 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*, pages 272–280, Washington D.C., USA, 2003.

[29] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, , and Darko Stefanovic. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, February 2005.

[30] GNU. The valgrind suite of tools for debugging and profiling linux programs. http://valgrind.org.

[31] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington DC, USA, August 2003.

[32] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. White Paper

http://www.research.avayalabs.com/project/libsafe/, December 1999.

[33] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of The 13th USENIX Security Symposium*, pages 45–56, San Diego, USA, August 2004.

[34] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows. To appear in Software—Practice & Experience.

[35] Solar Designer. Linux kernel patch from the openwall project. http://www.openwall.com/linux/README.

[36] grsecurity. Pax. http://pax.grsecurity.net/.

[37] Arjan van de Ven and Ingo Molnar. Execshield. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

[38] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[39] Peter Silberman and Richard Johnson. Attack vector test platform. http://www.idefense.com/iia/labs-software.php?show=1, February 2005.

[40] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[41] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[42] David A. Wheeler. Flawfinder. Web page http://www.dwheeler.com/flawfinder/, May 2001.

[43] Secure Software Soliutions. Rough auditing tool for security, RATS 1.3. http://www.securesw.com/rats/, September 2001.

[44] Alan DeKok. Pscan: A limited problem scanner for c source files. http://www.striker.ottawa.on.ca/~aland/pscan/, July 2000.

[45] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[46] Anup Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, May 1998.

[47] Wenliang Du and Aditya P. Mathur. Vulnerability testing of software system using fault injection. COAST, Purdue University, Technical Report 98-02 http://www.cerias.purdue.edu/coast/coast-library.html, April 1998.

[48] Janet Burge and Dave Brown. NFRs: Fact or fiction? Computer Science Technical Report, Worcester Polytechnic Institute, WPI-CS-TR-02-01 ftp://ftp.cs.wpi.edu/pub/techreports/pdf/02-01.pdf, November 2002.

[49] Lawrence Chung, Brian A. Nixon, and Eric Yu. Using quality requirements to systematically develop quality software. In *Proceedings of the Fourth International Conference on Software Quality*, McLean, VA, USA, October 1994.

[50] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[51] Premkumar T. Devanbu and Stuart Stubblebine. Security and software engineering: A roadmap. In *Proceedings of the Twenty-second International Conference on Software Engineering, ICSE*, Limerick, Ireland, June 2000.

[52] IEEE. IEEE-STD 610.12-1990, IEEE standard glossary of software engineering terminology, May 1990.

[53] Richard H. Thayer and Merlin Dorfman. *Software Requirements Engineering, Second Edition*. IEEE Computer Society Press and John Wiley & Sons, Inc., 1999.

[54] National Institute of Standards and Technology. Common criteria for information technology security evaluation (CC 2.1). `http://csrc.nist.gov/cc/CC-v2.1.html`.

[55] Herbert H. Thompson and James A. Whittaker. Testing for software security. *Dr. Dobb's Journal*, 27(11):24–32, November 2002.

[56] Microsoft. Microsoft security glossary. `http://www.microsoft.com/security/glossary.mspx`, November 2004.

[57] Robert W. Shirey. Request for comments: 2828, Internet security glossary. `http://www.faqs.org/rfcs/rfc2828.html`, May 2000.

[58] European Union. Common procurement vocabulary. `http://europa.eu.int/scadplus/leg/en/lvb/l22008.htm`, 2004.

[59] Mercell AB Sweden. Database of all purchases in the category '72 - computer and related services' made by swedish government or local authorities from January 2003 to June 2004. `http://www.mercell.com`, 2004.

[60] CERT Coordination Center. Cert/cc statistics 1988-2001. `http://www.cert.org/stats/`, February 2002.

[61] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[62] Frederick Giasson. Memory layout in program execution. http://www.decatomb.com/articles/memorylayout.txt, October 2001.

[63] 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 and Expo (DISCEX)*, pages 119–129, Hilton Head, South Carolina, January 2000.

[64] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.

[65] Aleph One. Smashing the stack for fun and profit. http://immunix.org/StackGuard/profit.html, November 1996.

[66] Gary McGraw and John Viega. An analysis of how buffer overflow attacks work. IBM developerWorks: Security: Security articles http://www-106.ibm.com/developerworks/security/library/smash.html?dwzone=security, March 2000.

[67] Matt Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, January 1999.

[68] DilDog. The tao of Windows buffer overflow. http://www.cultdeadcow.com/cDc_files/cDc-351/, April 1998.

[69] Lawrence R. Halme and R. Kenneth Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. http://www.sans.org/newlook/resources/IDFAQ/aint.htm, April 2000.

[70] Bulba and Kil3r. Bypassing StackGuard and StackShield. Phrack Magazine Volume 10, Issue 56 http://www.phrack.org/phrack/56/p56-0x05, May 2000.

[71] Crispin Cowan. Personal communication, February 2002.

[72] Istvan Simon. A comparative analysis of methods of defense against buffer overflow attacks. `http://www.mcs.csuhayward.edu/~simon/security/boflo.html`, January 2001.

[73] Mike Shuey Mike Frantzen. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[74] George Necula, Scott McPeak, and Wes Weimer. Taming C pointers. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, June 2002.

[75] George Necula, Scott McPeak, and Wes Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[76] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[77] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, May 2001.

[78] Wirex Crispin Cowan. Nearly 100 hackers fail to crack wirex immunix server, August 2002.

[79] Pierre-Alain Fayolle and Vincent Glaume. A buffer overflow study, attacks & defenses. `http://www.enseirb.fr/~glaume/indexen.html`, March 2002.

[80] David A. Wheeler. Secure programming for Linux and Unix HOWTO v2.89. `http://www.dwheeler.com/secure-programs/`, October 2001.

[81] tf8. Bugtraq id 1387, Wu-Ftpd remote format string stack overwrite vulnerability. http://www.securityfocus.com/bid/1387, June 2000.

[82] Scut and Team Teso. Exploiting format string vulnerabilities. http://teso.scene.at/articles/formatstring/, September 2001.

[83] Tim Newsham. Format string attacks. White Paper http://www.guardent.com/rd_whtpr_formatNewsham.html, September 2000.

[84] 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 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[85] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, Spring 1996.

[86] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[87] C E Pramode and C E Gopakumar. Static checking of C programs with LCLint. *Linux Gazette*, 51 http://www.linuxgazette.com/issue51/pramode.html, March 2000.

[88] S. C. Johnson. Lint, a C program checker. AT&T Bell Laboratories: Murray Hill, NJ. http://citeseer.nj.nec.com/johnson78lint.html, July 1978.

[89] Jose Nazario. Project pedantic—source code analysis tool(s). http://pedantic.sourceforge.net/, March 2002.

[90] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, `http://www.cs.berkeley.edu/~ushankar/`, August 2001.

[91] Jose Nazario. Source code scanners for better code. The Linux Journal `http://www.linuxjournal.com/article.php?sid=5673`, January 2002.

[92] Pete Broadwell and Emil Ong. A comparison of static analysis and fault injection techniques for developing robust system services. Technical report, Computer Science Division, University of California, Berkeley, `http://www.cs.berkeley.edu/~pbwell/saswifi.pdf`, May 2002.

[93] J S Foster, R Johnson, J Kodumal, T Terauchi, U Shankar, K Talwar, D Wagner, A Aiken, M Elsman, and C Harrelson. Cqual: A tool for adding type qualifiers to C. `http://www.cs.umd.edu/~jfoster/cqual/`, 2003.

[94] W Chen, B Rudiak-Gould, and B Schwartz. Automatic detection of implicit type cast errors in C. Paper in graduate course, `http://www.cs.berkeley.edu/~wychen/papers/261.ps`, 2002.

[95] R Johnson and D Wagner. Checking linux kernel user-space pointer handling with cqual. Work-in-progress report at IEEE Symposium on Security and Privacy, May 2003.

[96] K Ashcraft and D Engler. Using programmer written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[97] H Chen and D Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington DC, USA, 2002.

[98] V B Livshits and M S Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the*

11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Helsinki, Finland, 2003.

[99] Steven S Muchnick. *Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[100] M Weber, V Shah, and C Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[101] B V Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[102] B V Chess. Personal communication, 2004.

[103] Grammatech Inc. Codesurfer. `http://www.grammatech.com/products/codesurfer/`.

[104] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[105] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[106] M Musuvathi and D Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of the Second Workshop on Software Model Checking*, Boulder, Colorado, USA, 2003.

[107] K J Ottenstein and L M Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177—184, Pittsburg, Pennsylvania, 1984.

[108] N Walkinshaw, M Wood, and M Roper. The java system depencence graph. In *Proceedings of the Third IEEE International Workshop on*

Source Code Analysis and Manipulation, Amsterdam, The Netherlands, 2003.

[109] S Horwitz, T Reps, and D Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[110] J Ferrante, K J Ottenstein, and J D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[111] M Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, USA, 1981.

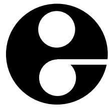
[112] T Reps and G Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, Washington DC, USA, 1995.

[113] Blexim. Basic integer overflows. Phrack Magazine 60 `http://www.phrack.org/phrack/60/p60-0x0a`, 2002.

[114] M Howard. Reviewing code for integer manipulation vulnerabilities. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp`, April 2003.

[115] David A. Wheeler. Secure programming for Linux and Unix HOWTO v3.010. `http://www.dwheeler.com/secure-programs/`, March 2003.

[116] Pia Fåk. Modeling and pattern matching security properties with dependence graphs. Master's thesis, Linkopings universitet, August 2005.



Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om *Linköping University Electronic Press* se förlagets hemsida `http://www.ep.liu.se/`

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the *Linköping University Electronic Press* and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: `http://www.ep.liu.se/`

© John Wilander
Linköping, 25th November 2005