國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

阻擋緩衝區溢出攻擊之自動化弱點保護系統研究

Automated Vulnerability-Protecting System to Practicably Defend against Buffer Overflow Attacks

研 究 生：張永錠

指導教授：張瑞川　教授

中 華 民 國　九 十 五　年 六 月

阻擋緩衝區溢出攻擊之自動化弱點保護系統研究

學生：張永錠　　指導教授：張瑞川 教授

國立交通大學資訊科學與工程研究所

摘 要

隨著網路發展日漸普及，人們依賴網路的程度也越來越高，因此網路安全問題開始受到重視，而其中 buffer overflow attack 就是重要的問題之一。過去 buffer overflow 的研究，有過高的 overhead 而難以實際使用或無法全面防護缺點，或是讓攻擊者仍有機會略過偵測而達成 buffer overflow 攻擊，進而讓攻擊者成功入侵並能完全控制系統。

我們提出一個針對軟體 buffer overflow 弱點部份進行保護的自動化系統，以有效防止攻擊者利用 buffer overflow attack 入侵控制系統。我們提出的自動化系統有如下特點，(1)我們的系統在偵測攻擊與辨識攻擊資訊之後，能夠利用這些資訊特別保護弱點區域；(2)能夠全面辨識程式中的攻擊資訊；(3)利用 ASR 並避免其缺點－guessing attack。根據實驗顯示，在系統遭受攻擊前，能夠隨時保持自動偵測攻擊的狀態，並擁有極低的 run-time overhead。而在偵測到攻擊之後，僅需 10%以內的 run-time overhead 來保護系統，使我們的機制能夠被實際使用。

Automated Vulnerability-Protecting System to Practicably Defend against Buffer Overflow Attacks protocols

Student: Yung-Ting Chang **Advisor: Dr. Ruei-Chuan Chang**

Computer Science and Engineering College of Computer Science

National Chiao Tung University

Abstract

Developing with the network popularity day by day, the degree of network reliance becomes higher. That results in more attention to network security. Buffer overflow is one of the most important problems in network security. Previous researches of buffer overflow defending have some shortcomings, which include high run-time overhead, defending only a part of program regions, or being bypass by attacker and giving attackers a chance to intrude and control the system.

We purpose an automated system to practicably defend against buffer overflow by protecting the region of the vulnerability of software. Our approach has following features: (1)After detecting the attack and identifying attack information, we can use the information to protect the region of the vulnerability specifically. (2)We can identify the attack information of the whole software regions. (3)We use ASR and eliminate the shortcoming－guessing attack. According to the experimental results, the system almost has no overhead before being attack and the overheads are lower than 10% after detecting attack. The results show that the performance is acceptable and make our approach practicable.

誌　　謝

首先感謝我最尊敬的指導老師 張瑞川教授。這兩年在老師費心的教導下，學生方能順利完成此篇論文。於授業期間，老師耐心的指導我正確的研究態度與研究方法，讓我受益良多。在撰寫論文期間，感謝大緯學長給予的建議以及不斷和我的討論。也感謝在實驗室一起努力的同學們，學長以及學弟們的支持和勉勵。

最後將此論文獻給我親愛的父母親。感謝您在我求學期間全心全意的支持，讓我得以專心的完成研究。

Index

LIST OF FIGURES

LIST OF TABLES

1 Introduction

隨著網路發展日漸普及，各式各樣的網路服務也越來越蓬勃發展，人們依賴網路的程度也越來越高。因此，網路安全問題開始受到重視。在網路服務軟體在設計過程中，程式設計者若稍一考慮不周，軟體就可能會產生漏洞，以致於在提供服務期間遭受攻擊。而其中最常見的攻擊型態之一即為buffer overflow attack。如Table 1，根據Computer Emergency Readiness Team(CERT ® Coodination Center) [4]軟體漏洞資料統計 [5]，每年軟體漏洞數量的成長率約為 50%，而其中光是會造成buffer overflow attack之漏洞就佔了所有軟體漏洞的 50% [43]，因此可以知道防止buffer overflow的重要性。

Year	**1995**	**1996**	**1997**	**1998**	**1999**
Vulnerabilities	**171**	**345**	**311**	**262**	**417**

Year	**2000**	**2001**	**2002**	**2003**	**2004**	**2005**
Vulnerabilities	**1090**	**2437**	**4129**	**3784**	**3780**	**5990**

Table 1: Vulnerabilities reported by CERT [8]

由於 C 語言本身設計上並不會自動檢查指標與陣列的使用邊界，導致寫入陣列時可能發生溢出寫入而改寫其他資料。這讓攻擊者有機會對有漏洞的網路服務軟體進行 buffer overflow 攻擊，這些網路服務軟體的權限通常都為系統管理者或 root，攻擊者就能利用含有惡意程式碼的 network input 作為溢出寫入的資料，服務軟體就會以 root 權限執行這些惡意程式碼，進而讓攻擊者入侵並控制此系統。

1.1 Overview of Buffer Overflow Protection

從Buffer overflow漏洞被發現[31]開始，buffer overflow防護機制的研究至今已十多年。部份研究專注於漏洞偵測。在此方面，一部份的研究在程式編譯時靜態分析程式碼中不安全的部份以在執行之前提醒程式設計者開發更安全的程式。然而，靜態分析通常無法完全偵測出漏洞，或者會出現過多的假警告。於是，大多數的研究利用動態偵測，在程式執行時偵測buffer overflow攻擊。這些包括偵測程式部分區域如return address[9][10]，利用address space[3][26]或instruction randomization[2][17]的方式，以及在程式中插入boundary checking程式碼[16][24]。只偵測部分區域讓攻擊者仍有機會攻擊程

式其他部分，randomization讓攻擊者能夠利用大量的猜測而攻擊成功，有些boundary checking的方式對所有程式中的buffer進行範圍檢查，卻產生了過高的overhead。而有其他相關研究，包括non-executable buffer[11][26][40]、monitoring control flow[18]、或是結合動態與靜態分析來偵測buffer overflow攻擊[15][23]，但這些偵測機制仍過高的overhead而難以實際使用，或是讓攻擊者仍有機會能夠成功攻擊而控制系統的缺點。

除了偵測buffer overflow攻擊外，近幾年的相關研究也強調在軟體遭受攻擊後，能自動辨識軟體漏洞[20][44]，自動產生network input signature[14][20][21][22][44]，或是自動修復服務軟體系統[27][39]。部分自動化系統利用過去的偵測方式，因此無法達到全面的保護或有過高的overhead而難以實際使用；自動產生signature的方式也不夠完善，無法產生準確的signature。錯誤的signature會導致正常的clients被系統拒絕而讓他們無法使用該網路服務，更重要的，它會讓攻擊者仍有機會略過而達成buffer overflow攻擊，進而讓攻擊者成功入侵並能完全控制系統。

我們提出一個自動化的系統，以針對軟體 buffer overflow 弱點部份進行保護。在尚未得知軟體弱點之前能夠隨時偵測是否有攻擊發生，當遭受攻擊時能夠偵測出攻擊資訊，並利用此資訊進行弱點保護，以有效防止攻擊者利用 buffer overflow attack 入侵控制系統。

1.2 Automatic Detection and Protection from Buffer Overflow Attack

我們提出的自動化系統有如下特點，(1)我們的系統在偵測攻擊與辨識攻擊資訊之後，能夠利用這些資訊特別保護弱點區域。為了能防護所有的buffer overflow攻擊並有極低的run-time overhead，我們採用Address Space Randomization (ASR) [26]來偵測攻擊，能鑑別遭受攻擊的buffer與overwrite該buffer的指令位址，並在自動偵測與鑑識軟體漏洞後，能夠快速自動防護該漏洞，相較於產生可能有false alarm的signature，我們能夠直接排除該漏洞，並自動針對該漏洞進行系統保護，因此不會有false alarm發生；(2)能夠全面辨識程式中的攻擊資訊。我們稍微修改[20][44]的方式，讓程式中動態配置的資料遭攻擊時也能夠辨識出以取得相關攻擊資訊；(3) 儘管利用ASR也能避免其缺點－guessing attack。讓我們的系統能夠保護control-sensitive data而讓攻擊者無法藉由大量的猜測而成功改寫此data，使修補後的系統能夠確實防止buffer overflow攻擊以避免攻擊者成功入侵系統。

我們在 Linux kernel 2.6.14 上將我們的自動化系統實作成一個 kernel module。在實驗中，我們利用 Apache、Sendmail、Thttpd、Telnetd、Wu-ftpd 等網路服務軟體，保護

這些軟體某些特定版本的弱點，測得我們的自動化系統能夠擁有很好的效能表現，驗證了我們的機制所產生之效能損失，是可以被接受而得以被實際使用的。根據實驗顯示，在系統遭受攻擊前，能夠隨時保持自動偵測攻擊的狀態，並擁有幾乎為 0 的 run-time overhead。而在偵測到攻擊之後，僅需 10%以內的 run-time overhead 來保護系統，使我們的方案能夠被實際使用。

1.3 Organization of this Paper

此篇論文剩餘部分的編排如下所述：第二章簡介 buffer overflow 攻擊以及討論過去的 related work，我們自動化系統的設計與實做將在第三章中描述，第四章將列出並分析我們的實驗結果。最後，我們於第五章做出此篇論文之結論。

2 Background and Related Work

我們在此章中先簡介 buffer overflow 之攻擊原理，並在之後介紹防範 buffer overflow 攻擊的相關研究。

2.1 Background: Overview of Buffer Overflow

由於 C 語言本身設計上的不安全與普遍使用，導致 buffer overflow 時常發生，當使用指標與陣列時，C 語言本身設計上並不會自動檢查邊界，而需程式設計者自行在程式中檢查之，再加上 C standard library 中的一些 string function 也是不安全的，因此若在撰寫程式時稍有不慎，就可能有 buffer overflow 漏洞發生。

Buffer overflow 的攻擊方法大致可由 Figure 1 解釋之，當程式進入 *foo( )*時，若 **sprintf( )**沒有做好邊界檢查，可能導致字串寫入 *buf* 陣列時超過其邊界，而由於 stack 的生長方向與字串寫入方向是相反的，則會 overwrite 進入方程式時 push 至堆疊的返回位址(***ret-address***)，當 *foo( )*返回時，processor 會使用 stack 中被改寫過的 ***ret-address*** 作為返回位址，這將使程式執行流程改變，返回到被改寫的位址上繼續執行。

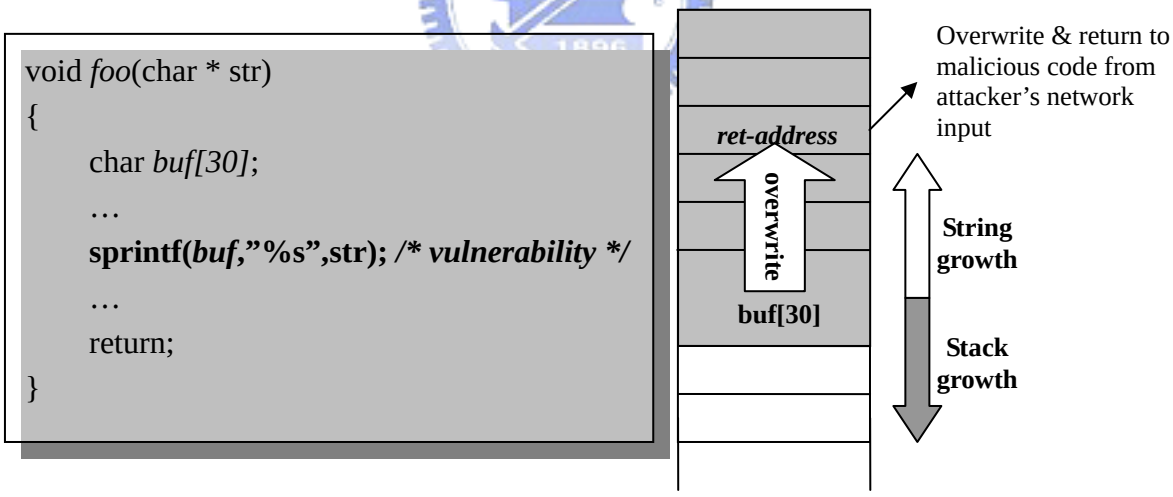


Figure 1: Typical scenario of buffer overflow attack

通常攻擊者會對使用有漏洞的網路服務軟體作為伺服器的機器進行 buffer overflow 攻擊，這些網路服務軟體的權限通常都系統管理者或 root，攻擊者會把他希望執行的惡意程式碼作為 network input，而在 ***ret-address*** 處填入這些惡意程式碼的起始點，因此由 *foo( )*返回時，就會以 root 權限執行這些惡意程式碼，完全控制此系統。這表示一旦被

攻擊者成功改寫控制程式流程的指標(如 return address 或 function pointer，這類資料稱為 control-sensitive data)，就代表了系統遭受控制，由此可知保護 control-sensitive data 的重要性。

2.2 Buffer Overflow Detector

Buffer overflow detector大致可以簡單的以static detector和dynamic detector分成兩類。Static detector [12][19][43][15][23]在compile time時判斷是否array或pointer有遭受攻擊的可能，若可能被攻擊，則detector就會有警告顯示。然而這樣的方法有許多的缺點[1][28]，無法偵測到所有的程式漏洞、會出現許多false positive warning，越注意所有遭受攻擊可能的工具越會顯示出許多假的警告；而為了降低顯示假警告的機會，則又反而可能會遺漏一些bug而讓攻擊者有機會入侵系統。再者，這些顯示出來的警告必須由programmer自行一個一個的檢查是否真的有問題，這些缺點讓static detector變得難以實際使用，因此許多的研究都朝向dynamic detector發展。

2.2.1 Detecting a Part of Program Regions

Dynamic detector能夠在攻擊發生時自動偵測之，有些buffer overflow detector，雖然overhead較低，但僅只能保護軟體中的部份區域，例如return address、function pointer、format string等。StackGuard[9][10]修改gcc編譯器，讓編譯過後的程式在執行至呼叫函數時能夠在stack中的return address之後置入一個canary value，這讓return address被overwrite時，canary value也會被overwrite，並在該函數返回時，檢察此canary value是否被修改，就能夠偵測出buffer overflow。Hiroaki Etoh等人[13]利用canary value以及重新排列stack上的陣列來保護return address與local variables。RAD[6]與StackShield[42]亦是修改編譯器，在推入return address至stack時，同時將這return address複製到另一處以利之後辨別比對此值是否遭修改。Libsafe與Libverify[1]在程式執行時動態載入他們修改的library，讓程式在起始時先修改每個函數，使這些函數能夠在被呼叫時預先儲存return address，再由函數返回時自動檢查return address是否被改寫而不需重新編譯程式。PointGuard[8]將指標以XOR加密，在需要時解密並reference該指標，讓攻擊者無法預測指標加密後的內容，但這樣只能保護到使用者程式的指標部分。FormatGuard[7]在call printf等function之前增加一個wrapper function，使其能在程式執行時偵測出攻擊者輸入的惡意字串。

2.2.2 Randomization

部分研究利用randomization的方式來使address、instruction不易被猜測而達到保護

效果，卻容易遭受guessing attack[36][41]，Address obfuscation[3]與Address Space Randomization (ASR)[26]隨機分配程式的heap、stack等分部位置使得攻擊者難以估算目標位址，使攻擊者在overwrite資料後因reference error而偵測出攻擊，而利用這樣的方式，能夠隨機搬移程式中所有區域的位置，因此能夠得到大範圍的保護，並且在執行期間不會花費額外的負載來偵測攻擊而不會有額外的overhead。Instruction Set Randomization (ISR)[2] [17]加密執行指令，並在每個指令執行前解密後由processor執行之，這讓攻擊者無法讓程式執行他們輸入的未加密過之惡意程式碼，但仍然會受guessing attack而使攻擊成功。

2.2.3 Bound Checking

有些detector能夠較大範圍或全面保護所有buffer overflow區域，但是overhead卻高達2倍至十幾倍，Robert等人[16]與Nicholas等人[24]皆是利用檢查指標指向範圍的方式，判斷是否於分配給它的記憶體範圍內，若超出範圍則表示可能有buffer overflow發生，但會產生false alarm。CRED[28]利用類似Robert的方法，但是調整了指標可接受的指向範圍，只有在指標參考了其指向位址時超出了邊界，才表示有buffer overflow發生，因為只要超出範圍的指標沒有參考該位址，也算是可接受的指標指向位址。

2.2.4 Other Issues

Non-executable buffer[11][26][40]將程式中的stack、heap區域更改為不可執行，這讓攻擊者無法執行他們輸入至此區域上的惡意程式碼，但攻擊者仍可利用返回至library的方式，並輸入他們的惡意參數，而達成攻擊的目的。Program shepherding[18]監視程式control flow，利用interpreter檢查每個branch之後的指令是否為原來的程式碼，或檢查呼叫library的指令位置是否正常，以防止執行惡意的程式碼，但也因此需要花費過大的overhead解譯每個指令所以難以實際使用。Cyclone[15]與CCured[23]結合static與dynamic分析，因為C是個不安全的語言，所以他們將它修改成較高安全度的語言，而static無法分析的部份則在程式執行時確認邊界來防止overflow發生，但是與C不相容的新語言卻讓大眾難以接受，。Zili Shao等人[37]同時利用硬體與軟體的方式，讓保護buffer overflow變的更有效率，而因為修改了硬體，牽扯到了相容性的問題，因此他們也認定這樣的方式只是適用在能夠依照不同需求而能讓軟硬體有特殊設計的Embedded System上，對於general purpose system，則難以實際使用他們的方式。

高 overhead 的 detector 在實際上並不適用，而高 performance 的 detector，或難以全面保護所有 buffer overflow 攻擊的部分，或有上述這些缺點，因而讓攻擊者仍得以成功

攻擊而控制系統。

2.3 Automatic System

最近幾年出現了一些新的 approach，除了提供新的偵測 buffer overflow 攻擊方式之外，也希望能自動鑑識 application 漏洞，或自動產生 network input signature，或希望自動修復系統。

DIRA [39]採用RAD的方式，以偵測他們是否遭受buffer overflow 攻擊這些data，利用memory updates log來達到memory corruption instruction identification，並需重新編譯以轉換軟體source code，讓該軟體能夠以checkpoint的方式，在遭受攻擊後能夠recovery軟體。但是為了降低overhead，DIRA只有在全域變數被修改時才行checkpoint，而非每次收到封包時就行checkpoint，因此在收到封包後且未行checkpoint時，就無法recovery成功，更重要的問題是，由於其採用RAD的方式，因此不能夠攔住所有的buffer overflow攻擊，這些缺點讓DIRA難以實際使用。

Martin Rinard等人[27]，試著在軟體發生memory corruption後，忽略掉超出邊界的寫入，將超出邊界的寫入另外儲存並在讓程式能讀取這些寫入時，而讓軟體繼續執行，但由於這可能導致程式的buffer儲存了錯誤的資料，所以無法總是recovery成功，使程式某些部分會執行出錯，100%~400%的overhead更是讓其難以實際使用之。

Stelios Sidiroglou等人[38]希望做到在得知軟體漏洞後，自動產生patch來修補這個漏洞，但是他們亦只能用一些簡單的方式來修補，例如讓導致overflow的buffer改成動態配置資料，但這樣的方式，讓此塊buffer還是同樣會發生buffer overflow而能改寫能控制程式流程的資料，所以無法總是修補成功，而且利用這種offline的方式取得patch，較難及時修補漏洞。

ARBOR[22][21]利用讀取input的歷史紀錄，取得此時讀取封包函數的位置、長度及型態，並利用這些資訊產生signature來過濾attack input，但相同的讀取位置可能有不同型態輸入，導致過濾器認定能接受的讀取長度較大，但攻擊者卻不用超過此長度則能攻擊成功，因此無法成功過濾所有攻擊，在作者研究的實驗中就有無法過濾攻擊的情況發生，並且會有產生false positive的情況。

TaintCheck[25]利用network input是否污染記憶體中的資料，只要與network input運算後寫入至記憶體中的資料，就代表是被污染的資料，若被污染的資料被當作指標使用，則表示此input為攻擊，並利用被污染的指標內容值作為signature，與Jun Xu等人[44]相同，如此過於簡單的signature在使用至非ASCII的協定如DNS、SSL時，就很容易有false alarm產生，其偵測攻擊的方式也導致run time overhead達到十倍以上，而讓他難以實際

使用。

在[20]中，Zhenkai Liang等人利用ASR來偵測buffer overflow attack，因此有很低的overhead，但為了產生較不容易有false alarm發生的signature，需要針對每種service訂定input format specification。而且他們利用input size作為判斷是否為attack input的方法之一，也導致了攻擊者可以利用漸漸縮小input size來不斷攻擊ASR的缺點–guessing attack [36][41]，使得系統被攻擊成功。

Nebula[14]只簡單使用一個signature就能在Network layer達到過濾buffer overflow attack input的效果，但由於其只判斷input中是否含有數個可能為stack的位址，因此只能保護stack overflow。針對利用heap overflow的攻擊則無法成功過濾。並且需要依照每種protocol決定input payload的哪些部份能夠略過filter，否則會有很高的overhead及false positive。然而，即使依照不同的protocol定了哪些input是不需過濾的，也會出現如同TaintCheck的情況，遇到binary protocol時會有false positive發生。

自動化系統的自動修補方式，有上述dynamic detector的缺點，只能保護程式部分區域或是overhead過高，這些缺點讓自動修補系統難以使用。而由於自動產生signature的方式也仍然不夠完善，因此仍會讓攻擊者略過過濾器，使得我們難以防止最希望能夠避免的問題，而讓攻擊者成功入侵並控制系統，這也是buffer overflow攻擊最主要的目的，因此這樣的自動化系統仍然難以實際使用。

3 Design and Implementation

Figure 2 顯示出我們自動化系統的架構。為了保護應用程式，我們在程式執行時，利用detector隨時監測程式，而在程式遭受buffer overflow攻擊時，detector將會發現此惡意攻擊，並將…交給analyzer分析之。Analyzer在辨識出保護機制所需要的資訊後，(將利用這些資訊在程式進入弱點區域時保護之)，Memory access controller即利用這些資訊更改弱點區塊記憶體的存取權限。Protection handler則會在應用程式改寫弱點區塊時，能夠判斷此寫入是否為合法的寫入。當遇到不合法的寫入時，則表示遭受攻擊，此時Protection handler會保護弱點區塊以免遭改寫，以達到防護buffer overflow攻擊的效果。

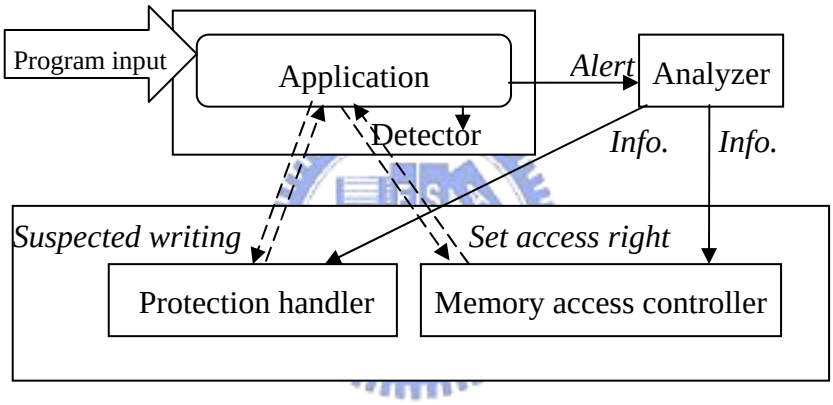


Figure 2: The architecture of our automatic protection

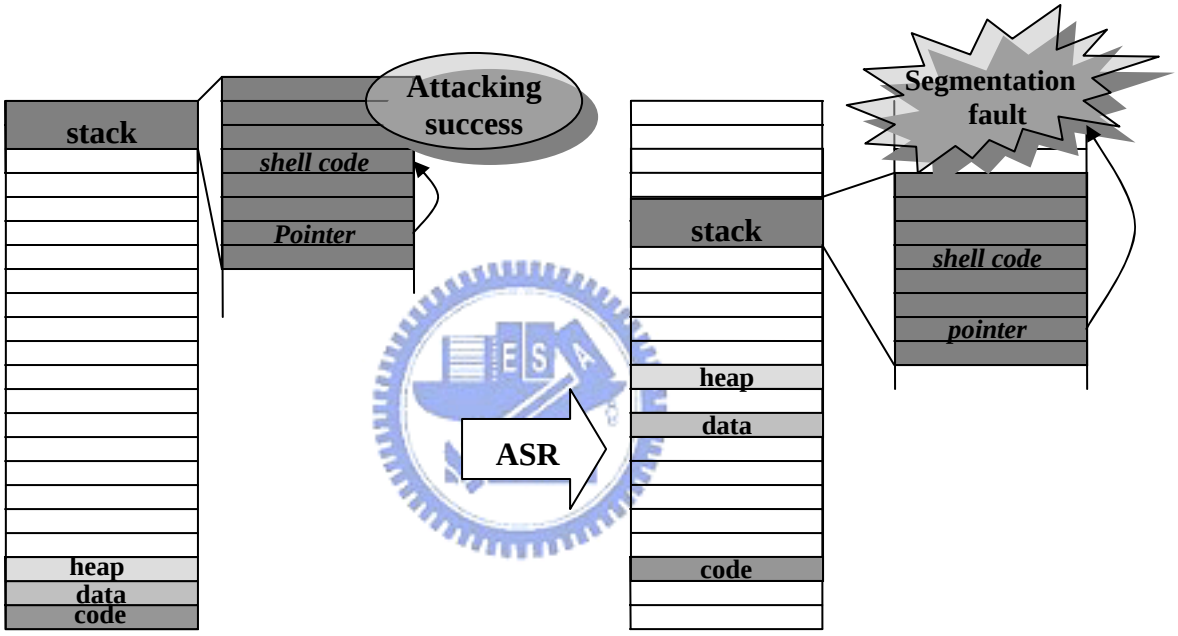
在以下幾個小節，我們將逐一介紹自動化保護系統每個部份的設計與實作方式，Detector負責buffer overflow detection，Analyzer處理偵測攻擊之後的identification，Memory access controller與Protection handler則利用attack information保護軟體弱點區域。

3.1 Buffer Overflow Detection

我們採用Address Space Randomization (ASR)[26]來達到偵測的效果，雖然目前有許多其他研究能夠偵測之並且這些方法也適用於我們的研究中，例如Instruction Set Randomization (ISR)[1][17]、StackGuard[9]、PointkGuard[8]、TaintCheck[25]，而如第二章所述，為了能有較低的overhead、有最寬廣的保護範圍，因此我們在實作時採用ASR，以達到偵測出buffer overflow attack的效果。

如Figure 3 所示，ASR將程式碼、shared library、stack、heap、static data等記憶體位

置隨機locate在address space上。當 attacker利用buffer overflow在overwrite 他想控制的pointer，如return address或function pointer時 (我們稱這些data為control-sensitive Data ，或簡稱CS Data)，他必須猜測該pointer需要的填入的內容值。但由於大部分的program只使用了所有address space的一小部分，所以attacker只有很小的機率[1]成功填入正確合法的address，這表示當reference或dereference該pointer時，會有很大的機會造成memory corruption，這時便成功偵測出攻擊。這樣的方式讓程式在執行時不會有如同其他detection方式需要花費額外的CPU時間來偵測寫入時是否超過buffer boundary，並且由於隨機配置程式所有記憶體區塊，而使其能夠大範圍的偵測所有buffer overflow攻擊。



Original address space layout　　ASR address space layout

Figure 3: The scheme of Address Space Randomization

我們採用 ASR 以偵測 buffer overflow，雖然這讓攻擊者只要一直不斷的猜測 pointer address，就會有機會攻擊成功。但它的低 overhead 與涵蓋的保護範圍廣泛，讓我們選擇他作為偵測方式，而其會被 guessing attack 攻擊的缺點也會在後面介紹我們產生的自動保護機制中得以被解決。

3.2 Previous Approaches on Buffer Overflow Identification

為了要防止程式被 buffer overflow 攻擊，我們必須在偵測到攻擊時，分析遭受攻擊

[1] 根據先前研究[36]指出，能夠攻擊成功的機率大約爲10^{-4}或更低。

的資訊，而為了保護會遭受惡意改寫的 buffer，我們必須先找出 CS data、程式執行到何處發生 process crash(稱為 *faulting instruction*)，以及該 CS data 是由何指令所改寫(稱作 *corrupting instruction*)。

過去大多數buffer overflow的相關研究多著重在偵測攻擊，直到最近幾年才有人提出自動化的防護診斷分析(Automatic Diagnosis)。因為有了自動化的分析，才能夠快速取得有用的資訊讓安全防護人員或程式設計者能更快速的修補漏洞。我們採用Zhenkai Liang 與Jun Xu 等人[20][44]的方式，Identification機制如Figure 4，大致如下：由於利用ASR偵測攻擊，我們能夠在發生memory corruption時立刻得知攻擊發生。若此時的PC (program counter)不是造成access violation的invalid memory address，則可利用CR2 register及分析該PC上的assembly code來追溯出CS data，而該PC即為*faulting instruction*；若此時的PC就是invalid memory address，則表示是在branch後發生segmentation fault，若是由function pointer發生branch，則可直接利用stack pointer取得return address，其前一指令即為*faulting instruction*，此時的CS data就是此function pointer。若是return instruction造成的branch，則將所有的return instruction都設定為breakpoint並利用logged message作為input重新執行程式，當再次發生crash時，最後一次breakpoint記錄下來的return instruction即為*faulting instruction*，而return address就是CS data。類似上述的方式重新執行程式，觀察每次寫入CS data的instruction，則在程式crash後，最後一次寫入的指令就是 *corrupting instruction*。

由於上述的方法，我們只能得到 CS data 在當次程式執行時所在的位址，若是動態配置的資料，如 heap 上的 buffer，由於動態配置記憶體的位址都是程式執行時才配置，因此無法在每次執行時都直接視該位址為 CS data，而我們若是無法在執行時知道需要保護的記憶體位置，就無法觀察每次寫入 CS data 的 instruction 而得到 corrupting instruction。

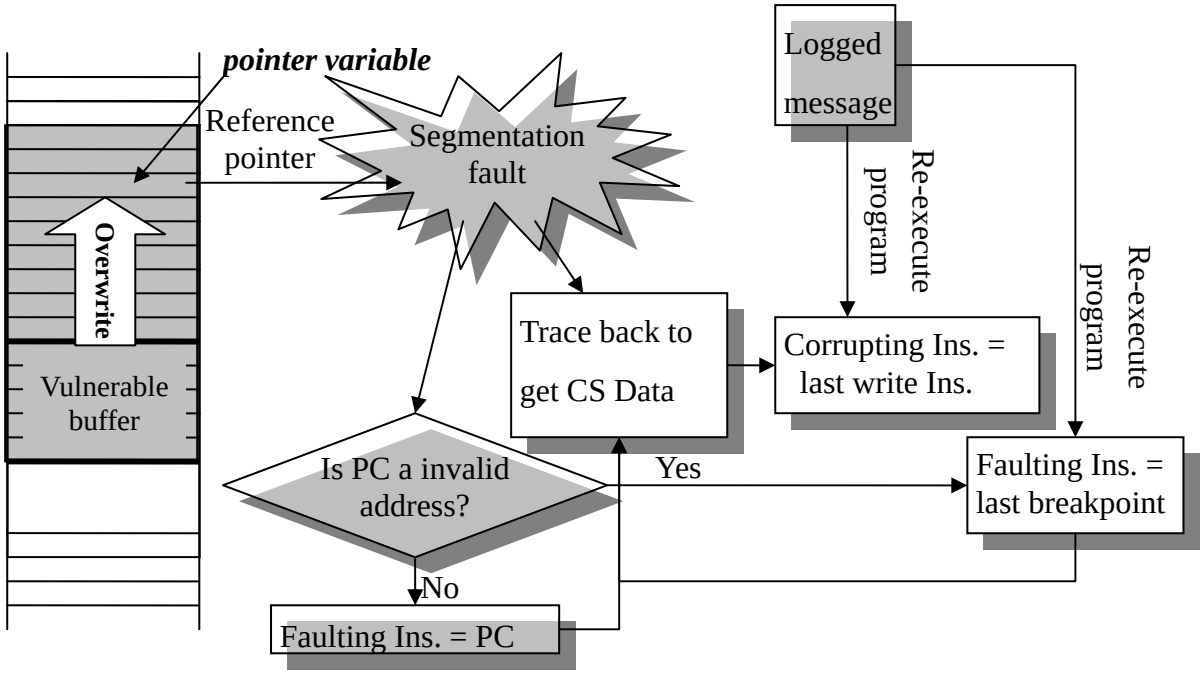


Figure 4: Identification of corruption instruction & control-sensitive data

3.3 Corrupting Instruction & Control-sensitive Data Identification

有鑑於此，為了能夠取得 CS data 為動態配置資料時的資訊，我們把上述的方法部份稍作修改，以符合我們的保護機制所需要的實際資訊。

3.3.1 Buffer Overflow Identification Suitable for Our Work

為了能夠辨識動態配置的資料，我們讓如 *malloc( )*、*calloc( )*等動態配置記憶體的相關函數外增加了一層 wrapper，使得程式在呼叫這些函數時，能夠將呼叫函數的位址以及配置的記憶體位置記錄下來，當偵測到攻擊並鑑識出 CS data 的位址後，就能夠根據記錄得知相對於該配置記憶體的函數呼叫位址。爾後程式執行時，若是由該位址之函數呼叫所配置的 buffer，就有可能是 CS data。除了記錄函數呼叫位址之外，我們同時記錄呼叫該函數當下的 call stack，符合 identificatoin 時的函數呼叫位址與 call stack，才有可能為 CS data。利用 call stack 作為辨識元素之一是因為許多程式會將 *malloc( )*等配置記憶體函數寫在自訂的函數中，若需要動態記憶體時，則呼叫自訂的函數，若是只利用函數的呼叫位址辨識是否為 CS data，將導致我們將一些非必要的資料視為須要保護的資料，而產生額外的 overhead。而由於是於 *malloc( )*等函數上加上 wrapper function，是屬於修改 shared library 的部份，只需要替換原來的 libc function 即可，不需重新編譯我們所要保護的程式。

若 CS data 為 local variable，同樣的，為了能夠在之後程式執行時保護該 data，我

們紀錄 identification 時該 data 位址與 stack frame pointer 的 offset，這樣就能在每次呼叫漏洞函數時，利用此 offset 取得當下的實際位址而保護之。

由於攻擊者在第一次攻擊失敗後，需要修改填入 CS data 的位址並再次攻擊以達到控制弱點機器目的，因此我們在實作上能夠利用等待再次的攻擊取代 logged message 以辨識這些資訊，這可知最多只需再額外等待辨識 faulting instruction 與 corrupting instruction 所需的兩次惡意輸入，就能夠取得所有有用資訊。若攻擊者想對特定 server 進行攻擊，從一次變成三次的機會要猜測出所需的正確位址，也仍然非常困難，因此我們能夠採用這樣的方式取代原本的 logged message。

由於我們採用 ASR 來偵測攻擊，當程式重新執行時，程式中的 corrupting instruction 位址也會更動，因此我們記錄偵測出 corrupting instruction 時，該指令與 code section 起始處的 offset，則在每次執行 program 時，就能夠利用已知的 offset 與 code section 起始位置求得實際的 corrupting instruction 位置。

利用上述的方式作 detection 與 identification，可以知道一般情況下，系統管理者在一開始通常會選用最新與最穩定的版本作為服務軟體，在服務啟動一段時間後才會有發現漏洞或遭受攻擊，此時該軟體開發者會盡快修補漏洞，以提供給使用此服務軟體的伺服器更新，這表示在大部分的時間下，也就是服務啟動後，在尚未發生任何漏洞或遭受攻擊之前，我們幾乎不用為了防止程式遭受 buffer overflow 攻擊而需額外的 overhead，而能夠讓軟體一直保持最高的效能持續服務客戶。

3.4 Automatic Protection from Buffer Overflow Attack

在偵測到攻擊並取得 identification 資訊後，我們重新啟動服務軟體，並啟動保護機制。我們已經知道 identification 後得知 corrupting instruction 的位址，由於程式漏洞，此指令會在不正當使用時改寫 CS data，而這些不正當的改寫都是發生在此指令所在的函數中，因此我們將此函數稱為 vulnerable function，只有進入此函數中，才有可能發生不合法的寫入，所以只需要在此時特別保護會被破壞的資料，其他時候則不需耗費任何資源保護，這樣可盡量將服務軟體的 overhead 降低，以避免效能損失在保護機制中。

3.4.1 Protecting control-sensitive data in vulnerable function

由於我們已經得知 CS data 資訊，在 identification 後，我們修改程式的 binary code，將 vulnerable function 的最前與最後一個指令取代成 breakpoint instruction (*int3*)並啟動程式，則在每次進入 vulnerable function 後，其概念大致可用 Figure 5 解釋，程式會在執行 *int3* 後馬上進入 int3 handler (memory access controller)，我們在此將該 data 所在的 page

設為唯讀，若發生不合法的寫入到 CS data，則代表有遭受攻擊，如此一來，一旦我們完成弱點辨識，則攻擊者就絕對無法攻擊成功，因為他只要一想要改寫關鍵資料，就馬上會被系統偵測出，利用這樣的方式保護弱點程式，能讓程式防止 CS data 遭受惡意改寫。

而因為我們利用 breakpoint instruction 將原本進入函數後的指令取代了，因此必須在 handler 內運作此指令，否則該函數將會因為沒有執行該指令而發生程式出錯，剛進入函數的第一個指令是將 stack frame pointer 堆疊至 stack 中(*"push %ebp"*)，由於此指令是在 user mode 中執行，但是進入 int3 handler 的時候已經是在 kernel mode 中了，此時我們使用的 stack 與所有暫存器都和在 user mode 的時候不同，因此我們將 user mode 的 stack pointer (%*esp*)減一個 word 並在此位址存入 user mode 之*%ebp* 內容值，以替代原本的*"push %ebp"*。

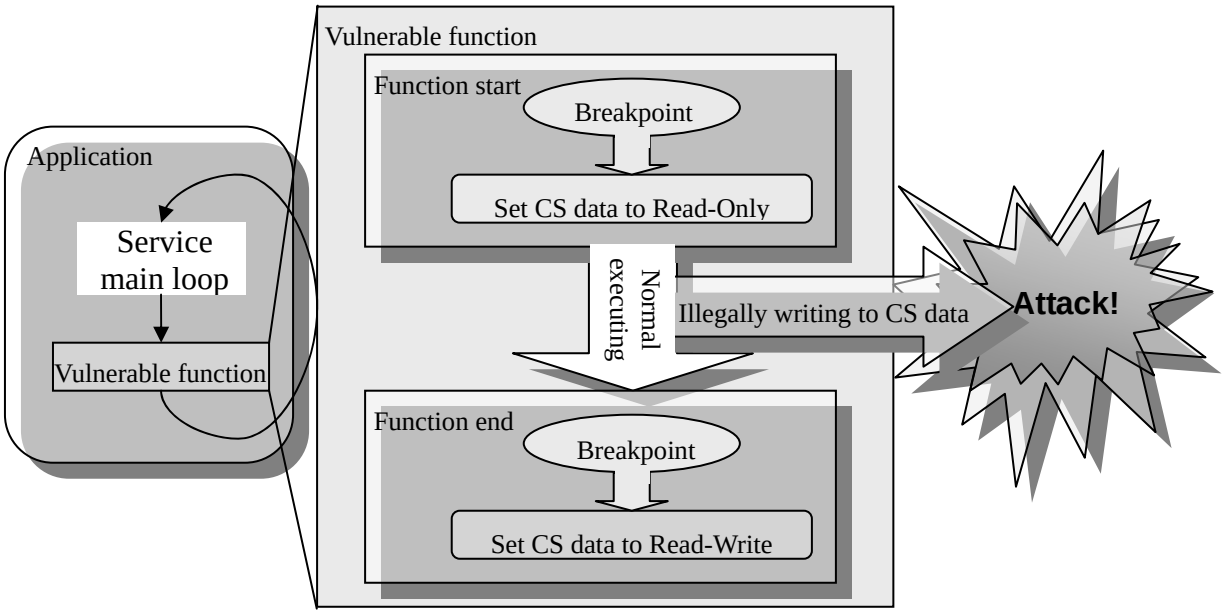


Figure 5: Protecting Control-sensitive data in vulnerable function

在保護 CS data 這段期間,若沒有惡意的改寫發生,則在 vulnerable function 要 return 前，再次進入 memory access controller，此時則將原本設定成唯讀的 page 改回可寫，在離開此函數後，就不會出現額外的負載。同樣的我們也必須在此處執行原本被取代的指令,而函數在返回之前執行的指令顯然是返回指令(*"ret"*),我們從 user mode stack pointer 中取得返回位址，將 user mode program counter 取代成此值，使得返回 user mode 時能夠直接執行程式返回時所要執行的位址，以替代原本的*"ret"*。

這樣的保護方式，和 ASR 在 CS data 被改惡意改寫之後，reference 該 data 才偵測

出不同，因為一旦攻擊者藉由大量的猜測而成功改寫 CS data 成正確的位址，ASR 將無法成功保護系統，我們在 CS data 被改寫之前即攔截惡意的輸入，以真正達到保護軟體漏洞的效果。

3.4.2 Extra handling of writing to non-CS data in the read-only page

由於我們利用分頁保護機制來讓寫入 CS data 時能夠產生 exception，才能在 exception 中處理 CS data 的保護，但是分頁保護機制是以一個 page 為單位，只要將一個分頁設定為唯讀，則程式在寫入此分頁中的任何位址，都會引發 page fault exception。然而，並不是只要一發生 page fault 就代表這時候遭遇了非法的寫入。當 exception 發生時，有兩種情況我們必須視為合法的寫入，(A) 寫入此分頁中的位址，但並非是寫入 CS data 所在的位址；(B) 寫入 CS data 但是為合法的指令寫入之，第二種情況我們將在下一節介紹之。在遇到合法寫入時，我們應讓弱點程式繼續正常執行。

由於我們在memory access controller中是直接改寫CS data所在page table entry，將entry中的W/R旗標改成唯讀，而在我們的protection handler判斷程式中的寫入指令為合法寫入之後， Linux kernel本身的page fault handler (*do_page_fault( )*)會檢查發生fault的分頁在kernel紀錄的memory region中是否擁有寫入權限，這時候kernel會發現此process是有寫入權限的，所以kernel會認為此分頁為demanded page或是copy-on-write (COW) page，並利用present flag判斷是兩者中其中之一[2]，若是demanded page則會在取回分頁時確實執行該寫入；若是COW時，kernel會判斷是否該分頁確實需要複製一份，若只有一個process擁有此分頁，則COW就不會真的起作用，並且會將此分頁的寫入權限打開。

當我們遇到合法的寫入時，必須在 page fault exception 結束後，再執行一次該寫入指令，而上述 kernel 將此 page 視為 COW page 時，就會在例外返回時再執行一次該指令，因此這與我們期望的相符，但這也讓 CS data 所在的分頁被改為可寫，在 vulnerable function 中若只因為一次的寫入到此分頁，之後就不能再利用此唯讀分頁監視寫入指令，則將無法達到確實保護 CS data 的效果。

因此我們必須在執行寫入指令之後，立即將該分頁再次設定為唯讀。我的方法是將寫入指令之後的下一個指令取代為 breakpoint instruction，使得寫入指令執行完之後能

[2] present flag代表該分頁存在與否，若此分頁不存在(present flag = 0)則表示此分頁不在RAM中，這有可能是因爲process尚未存取過該page而且此分頁是確實存在的但是暫存在硬碟中，這時kernel便需要將此page載入；若此分頁存在(present flag = 1)，則kernel就會認定此分頁爲copy-on-write，這代表該分頁可能是與其他process共用，但當需要寫入時則必須再複製一分頁，以免分頁被修改而影響其他process。

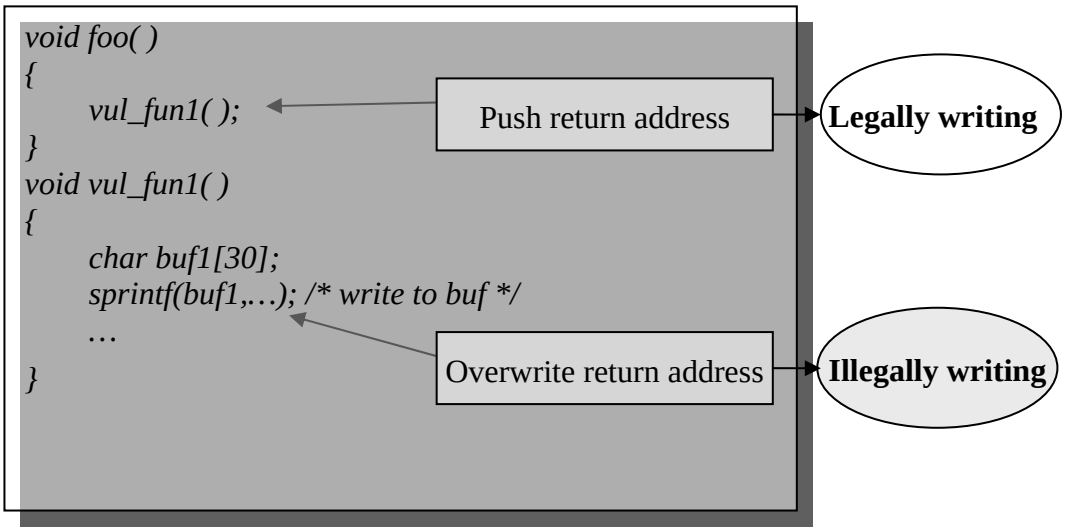
夠立即進入 int3 handler。當執行到此 handler 時再將需要保護的分頁設定為唯讀，把原本將其置換的指令存回原處，並且讓 user mode program counter 再次指向此被取代的指令，在返回 user mode 後，就能執行原來被取代的指令並繼續正常執行程式。

3.4.3 Differentiating Between Legal and Illegal Writes

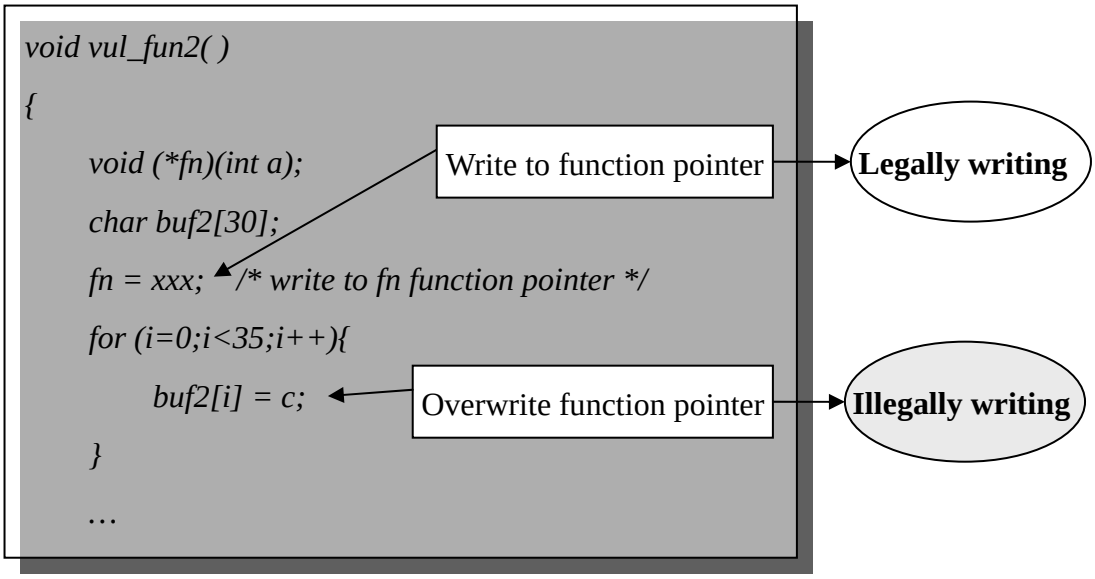
在正常情況下，寫入一般 buffer 卻惡意要竄改 CS data 的指令，雖然是個寫入的指令，但它並不會改寫到 CS data，這是因為 buffer overflow 有一個特性，就是利用原本要寫入 buffer 的指令，繼續在超出 buffer 邊界後寫入，使其能夠改寫 CS data。Control-sensitive data，包括 return address 與 function pointer 等，都是負責控制程式執行流程的指標，而被 overflow 的 buffer 則是不同類型的變數，因此程式並不會同樣的方式對這兩種變數進行寫入。如 Figure 6(a)，在 *foo( )*中呼叫 *vul_fun1( )*函數，這時候 return address 會被存入 stack 中，進入 *vul_fun1( )*後，由於沒做好邊界檢查，使得 *sprintf( )*在寫入 *buf1* 後，仍繼續寫入至原本儲存的 return address。我們可以發現，此時惡意的改寫指令為 *sprintf( )*中的寫入指令，與此 return address 的合法寫入指令，也就是呼叫 *vul_fun1( )* 函數的指令(*"call vul_fun1"*)不相同。另一個例子如 Figure 6(b)，在 *vul_fun2( )*中*"fn = xxx;"*將 *xxx* 寫入 *fn*，使得此 function pointer 能夠在被呼叫時跳至位址 *xxx*，但因為 for 迴圈中寫入 *buf2* 時超過邊界而改寫 *fn*，可知此例子的合法寫入 fn 為*"fn=xxx"*，而不合法的寫入則為 for 迴圈中的寫入指令，此兩個寫入指令是不相同的。

由於 buffer overflow 攻擊有上述這樣的特性，因此我們定義不合法的寫入必須同時符合以下兩者：(1) 在 vulnerable function 中改寫 CS data。(2)改寫指令位址為我們 identification 時得到的 corrupting instruction 位址。若在 protection handler 中發現改寫指令同時符合這兩個條件，則我們將把此此改寫指令視為不合法的寫入，這時後便能確定這是一個 buffer overflow 攻擊，而能及時防止其改寫 CS data。

如果攻擊者能夠用不同的 corrupting instruction 來攻擊，就表示這是程式中的其他漏洞，讓攻擊者能夠利用之，若攻擊者利用不同的漏洞攻擊系統，則同樣的會觸發 detector，進入 identification，最後同樣也能以我們的保護機制阻擋這個攻擊。



(a) Overwrite return address



(b) Overwrite function pointer

Figure 6: The difference between writing to CS data and writing to a buffer

我們認為攻擊者利用的 buffer 寫入指令，與正常 CS data 的寫入指令不相同，是因為 buffer overflow 攻擊上述的特性，因此可以知道這樣的方式讓攻擊者難以，若攻擊者真的成功利用原本就是作為寫入 CS data 的指令，來惡意改寫 CS data，則依照我們之前所述的 identification 方式，我們依然會認定此指令為不合法的寫入指令，此時會導致攻擊者與合法的使用者皆無法使用會進入 vulnerable function 的服務。但因為上述 buffer overflow 攻擊的特性，我們可以知道同時為 buffer 寫入指令與 CS data 寫入指令的機會

並不高，若確實發生了，此時正常使用者在進行不會進入 vulnerable function 的情況下，也仍然可以正常使用此服務；而面對攻擊者的輸入時，系統會認為這是不合法的寫入，仍然能夠阻擋攻擊者入侵系統，讓攻擊者無法達到 buffer overflow 的最主要目的。

在我們保護CS data時，若發現到惡意改寫，則代表程式中的其他資料也可能遭到改寫，此時我們將必需重新執行服務程式，以避免被改寫的其他資料使程式發生錯誤，這樣的方式與過去如related work所述的動態偵測buffer overflow攻擊之研究皆相同。自動化系統中，Martin Rinard等人[27]避免了超出buffer邊界的寫入，如related work所述，仍然導致程式可能有錯誤發生，而數倍或可能到達數十倍的overhead更是讓這樣的方案難以實際使用。自動產生signature的方式也仍然有缺陷，所以讓攻擊者能略過自動產生的signature，而發生最嚴重的情況，就是讓攻擊者成功入侵並控制系統，而這也是buffer overflow攻擊最主要的目的，因此這樣的自動化系統仍然難以實際使用，因此我們的研究最主要的目的，就是能夠成功防止攻擊者入侵控制系統，並且在執行其間，能夠有很好的效能表現，讓我們的研究能夠實際使用。

在遭受 buffer overflow 攻擊並辨識出所需的資訊之後，完整的保護機制可利用 Figure 7 解釋：程式在執行至 vulnerable function 之後，會在該函式一開始執行*"int3"*而進入 memory access controller，此時會把 CS data 所在的分頁設定為唯讀，並回到函數中繼續執行，若函數中的都沒有寫入至此唯讀分頁的指令，則會在函數要返回時執行*"int3"*而將此分頁的讀取權限改回可寫，之後程式便能繼續以正常狀態執行，若在函數返回之前，有任何指令改寫該唯讀分頁，則會進入 protection handler，此時會檢查該寫入指令是否是寫入 CS data，如果是的話會再判斷該寫入指令是否為 corrupting instruction，符合此兩個條件則代表不合法的寫入，若是不合法寫入則代表遭受攻擊；若不是寫入至 CS data 或不是 corrupting instruction 造成的寫入，則會將程式的下個指令取代為*"int3"*，並返回程式中執行，程式在完成寫入指令後馬上又進入 memory access controller，則會將原指令儲存回被 breakpoint 取代的位置，並再次把 CS data 所在之分頁設定為唯讀，如此反覆循環，即為我們防止 CS data 被惡意改寫的機制。

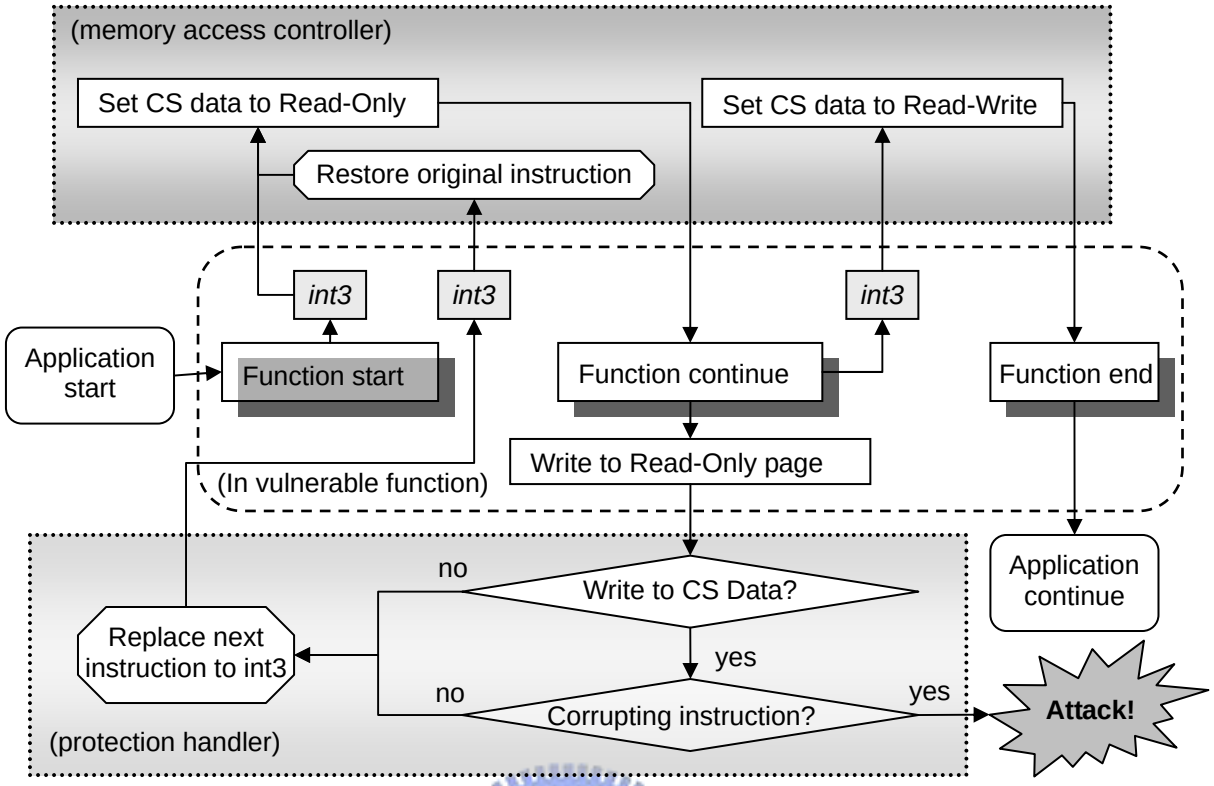
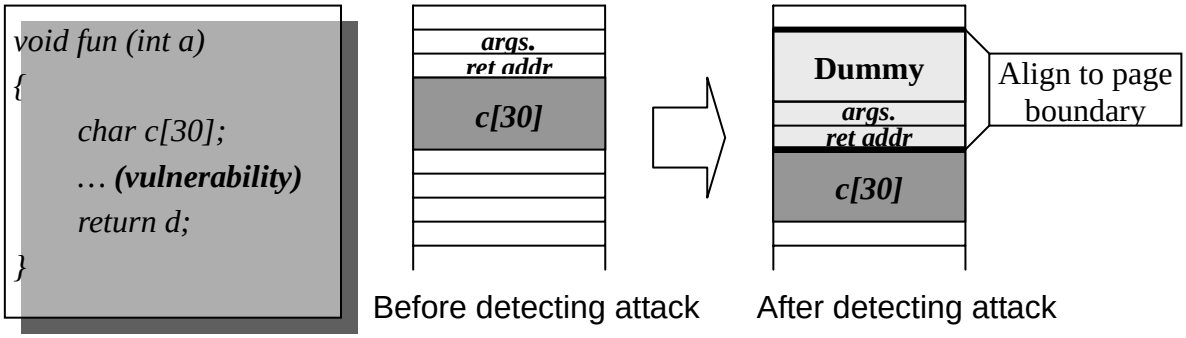


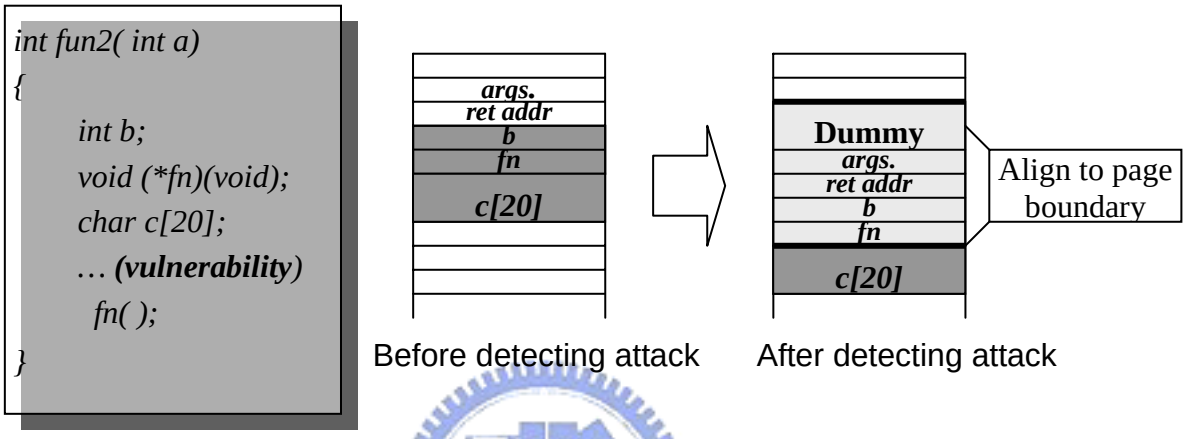
Figure 7: The scheme of control-sensitive data protection

3.4.4 Reducing Protection Overheads

為了保護 CS data，我們在必要時將其所在的分頁設定為唯讀，但是在程式執行寫入指令至該分頁時，就會進入 protection handler，並判斷是否為合法寫入，若是合法的寫入，則程式繼續執行，這代表這些保護處理，變成了額外的 overhead，因此我們採取以下方式，以降低進入 protection handler 的機會。



(a)When CS data is a return address.



(b)When CS data is a local variable.

Figure 8: Location of CS data in stack section

為了保護 CS data，我們在必要時將其所在的分頁設定為唯讀，但是在程式執行寫入指令至該分頁時，就會進入 protection handler，並判斷是否為合法寫入，若是合法的寫入，則程式繼續執行。這代表這些保護處理，變成了額外的 overhead，因此我們採取以下方式，以降低進入 protection handler 的機會。

當 CS data 為 return address 時，若我們直接將 return address 所在的分頁設定為唯讀，則在 vulnerable function 中呼叫其他函數導致堆疊(push to stack)或是對區域變數的進行寫入動作，只要仍在 stack pointer 仍在此唯讀分頁上，就會進入 protection handler 而產生大量的 overhead。因此我們利用如 Figure 8(a)的方式，*fun( )*的 ***vulnerability*** 處讓攻擊者有機會改寫 return address，當 CS data 為 return address 時，我們在一進入 vulnerable function 時，就把 stack pointer 指向的 return address 與此函數所需的參數往下挪動，讓 return address 與 page boundary 對齊，也就是讓 return address 放在唯讀分頁中的第一個位址，因此 vulnerable function 內所宣告的任何區域變數，包括其呼叫之函數的區域變數，都在 return address 所在的下一個或更後面的分頁，所以對這些變數的存取，都不在

唯讀分頁上，這避免了在函數中對自己的區域變數進行大量寫入的指令導致時常進入 protection handler，而從 vulnerable function 返回時，我們便在 int3 handler 中把 stack pointer 指回原來 return address 所儲存的位址，並且把該分頁改回可寫，所以在這之後又不會有 overhead 了，因此這樣的作法能夠盡量避免了不必要的 overhead。

若 CS data 為 return address 以外的其他區域變數，我們也利用類似上述方式(Figure 8(b))，*fun2( )*的 ***vulnerability*** 處讓攻擊者有機會改寫 *fn* function pointer，memory access controller 在進入 vulnerable function 時，先修改 stack pointer，讓 CS data 能夠儲存在分頁的最前面，雖然會讓 CS data 之上的區域變數也在唯讀分頁內，但仍然能夠排除 vulnerable function 中非唯讀分頁內的其他區域變數，以及 vulnerable function 中被呼叫的函數內所宣告的區域變數。

4 Experimentation and Observation

我們將我們的自動化系統實作成一個 kernel module。我們使用的系統平台為 Gentoo 2005.1，Linux kernel 版本為 2.6.14，作為伺服器的作業系統實驗環境。硬體配備為 Intel 3.2 GHz Pentium 4 CPU，512 MB DDR RAM。客戶端的作業系統版本為 Red Hat 8.0，Linux kernel 2.4.18，硬體配備為 Intel 1.6 GHz Pentium 4 CPU，256 MB DDR RAM。網路環境為 100Mbit/sec Ethernet。我們在伺服器上執行各種網路服務軟體，並在客戶端執行各種 benchmarks，以比較各種服務軟體在加上我們的保護後之效能表現差異。

4.1 Observation of legal write and illegal write to CS data

我們觀察了如 Table 2 中的 15 種知名的網路服務軟體所被公布出的實際弱點案例以驗證我們在第三章中所述的不合法寫入之定義。這些例子中包含改寫 return address、local data、heap data 與 global data。我們比較這些不同的 buffer overflow 例子中，攻擊者利用漏洞而使用的各種寫入 CS data 之指令，以及程式中原本正常寫入 CS data 之指令，經過驗證之後，確實發現這些實際軟體所被惡意改寫的部分，與合法的寫入動作是不同的。藉由這些例子，更加確認我們對不合法寫入的定義。

Application	version	vulnerability
Sendmail	8.12.9	*Prescan() vulnerability to overwrite to heap control structure*
Mysql	3.23.31	*Overrun return address when supplying SELECT statement*
Bind	8.2.2	*Exploit malloc's internal variables when handling transaction signatures*
Telnetd	0.17	*Overwriting setenv( ) library function pointer*
ProFtpd	1.2.7	*Overwriting 2 bytes to heap control sturcture*
Mysql	3.23.54	*Double free heap overflow to rewrite control structure*
Samba	3.0.8	*Overwrite next heap control structure*
Squid	2.5	*Return address overflow*
Thttpd	2.22	*Overwriting ebp in defang()*
Oracle	9.0.2	*Return address overflow*
Snort	1.9.x	*Overwrite function pointer (static data) at memcpy in TraverseFunc()*
Wu-ftpd	2.5.0	*mapped_path buffer without bound checking to overwrite longjmp data*
xinetd	2.1.8	*Return address overflow*
Apache	1.3.32	*Get_tag() vulnerability to overwrite return address*

Table 2: Observation of illegally writing to control-sensitive data

4.2 Overhead of Recording Heap Data

如第三章所述，為了能夠辨識出動態資料，我們在動態配置記憶體函數外增加額外的 wrapper。因此，在尚未偵測到攻擊之前仍然會有 overhead 存在。我們利用不同的網路服務軟體測量了此時的效能表現差異。

4.2.1 Sendmail

我們在伺服器上安裝 Sendmail version 8.12.9，並在客戶端執行 Postal benchmark，測量每分鐘伺服器所接收的 message 數目與資料量。此時採取我們方案的 sendmail 會在每次呼叫動配置記憶體函數時，進入 wrapper 以紀錄 heap data 資訊，而我們使用 Postal benchmark 能夠測量每分鐘 mail server 能夠接收的 message 數量與總資料量。Figure 9 為實驗結果，由圖中可知，原本的程式(Figure 9 中的 normal)與我們的方案只有些微的差距，包括所測得的 message 數量或接收的資料量，平均 overhead 都在 1%以內，可知我們的方案在此階段有著很好的效能表現。

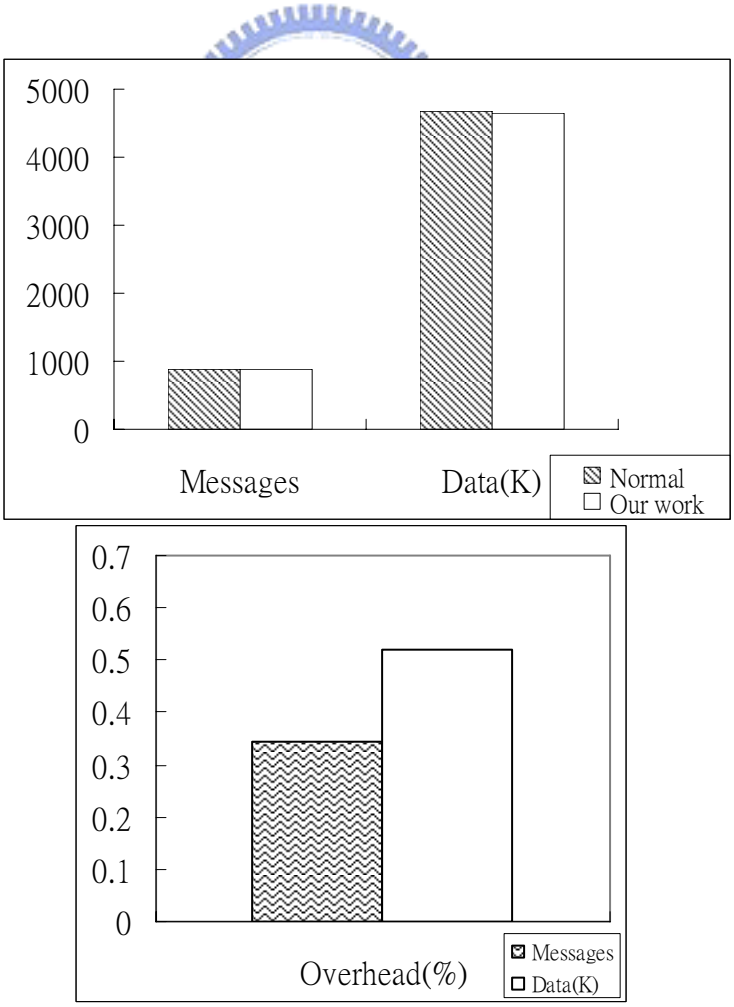


Figure 9: Performance and overhead on Sendmail of recording heap data

4.2.2 Apache

類似上述實驗，我們改在伺服器上安裝 Apache 1.3.32，並在客戶端執行 Webstone benchmark，測量 server throughput，比較原本的 Apache 與我們方案的 Apache 之表現差異，經過測量得到 server throughput 從原本的 74.61Mbits/sec 降至我們方案的 74.096 Mbits/sec，overhead 不到 1%。觀察後發現 Apache 並不會在每次 client 進行 request 時都會呼叫動態配置記憶體函數，因此我們方案的 wrapper 並不會經常被執行而導致可觀的效能下降，可知在此例子下，我們方案的效能表現是和原本是幾乎相同的。

4.2.3 Thttpd

同樣的我們利用 thttpd 2.22 作為 Web server，而利用 Webstone 在客戶端進行 request 以測量原本 thttpd 與我們方案之 thttpd 的效能表現差異，測量後發現 server throughput 都是 76.9 Mbits/sec，這是因為 thttpd 在 client 進行 request 不會呼叫動態配置記憶體函數，因此不會有 overhead 產生，而能夠和原本的 thttpd 有同樣的效能表現。

4.3 Overhead of Identification

由於我們在偵測到攻擊之後，如第三章所述，實作上是利用等待再次惡意輸入以辨識其他所需攻擊資訊的方式，因此在辨識攻擊資訊的這段期間，我們會有額外的效能損耗在辨識機制上。

4.3.1 Apache

以Apache 1.3.32 為例[30]，在攻擊者對Apache進行攻擊而發生segmentation fault而我們辨識出攻擊者所攻擊的Control-sensitive data為return address之後，如第三章所述，我們會把所有的return instruction都取代為breakpoint，而每次進入breakpoint handler時就會記錄此次的執行位址，此時儘管是合法的client進行request，我們也必須花費額外的負載，以辨識是由何函數的return instruction造成segmentation fault。我們利用Webstone benchmark，向此時會記錄每次執行return instruction的Apache進行request，以測量此時所的效能表現，由Webstone benchmark所測得發現server throughput會由原本的 62.6 Mb/sec降低為 60.4 Mb/sec，overhead為 3.7%，因此能夠知道，在此情況下的效能下降是能夠被接受的。

在辨識出 faulting instruction 之後，我們等待再次惡意輸入以辨識 corrupting instruction，同樣的我們利用上述的 Apache 實驗環境，測量辨識 corrupting instruction 所需的 overhead，在每次對 cs data 進行寫入時，都會進入我們的 protection handler 以記錄

此寫入指令而造成額外的 overhead，經過 Webstone benchmark 測量後，發現並沒有額外 overhead 產生，這是因為正常的 client 在進行 request 時，並不會對儲存在 stack 上的 return address 進行改寫，因此在這個情況下，我們利用 Webstone 所產生的所有的請求，都不會有 overhead 發生，可知道在此情況下，我們等待再次惡意輸入來辨識 corrupting instruction 的方式，是不會讓合法的請求產生效能下降的。

4.3.2 sendmail

我們再以Sendmail version 8.12.9 為例[33]，測量偵測到攻擊與辨識出攻擊者欲竄改的cs data之後，server等待再次惡意攻擊以辨識corrupting instruction時的效能表現差異。我們在客戶端執行Postal benchmark，發現測得的message數量由每分鐘 947.4 降至 868.6，而每分鐘接收的資料量由 5001.6K降至 4638K，overhead分別為 8.31%與 7.27%，可以知道此時的效能下降仍是可以被接受的，這表示在此情況下，我們在實作上利用等待再次惡意輸入以辨識其他所需攻擊資訊的方式，是能夠被實際使用的。

4.4 Overhead of Protection

由第三章所述的保護機制可知，我們在 vulnerable function 中檢查不合法的寫入，由於進入 vulnerable function 的機會，與在 vulnerable function 中寫入 CS data 所在的分頁中的機會，會依照不同的程式漏洞而有所不同，而產生不同的額外負載比例，因此我們在接下來的實驗中，分別測量 Apache、Sendmail、Thttpd、Telnetd、Wu-ftpd，以觀察我們的保護機制所產生的效能差異。這一節的實驗是測量在偵測到攻擊與 identification 後的各種 service 效能表現，也就是在已經獲得攻擊者欲改寫之 CS data 以及 vulnerable function 等資訊之後，我們將各個 service 中欲保護的函數之起始與結束指令取代成 breakpoint，之後即開始我們的測量，利用 benchmark 或測試程式對各種 service 進行 request，當 request 造成 service 進入 vulnerable function 時，會執行 breakpoint 進入我們的 handler 以及利用已取得的攻擊資訊進行保護處理。可知道這時候的情況變會與原本的 service 產生效能差異，接下來將介紹各種不同的服務軟體與攻擊所產生之實驗結果。

4.4.1 Apache

此實驗的Apache版本為 1.3.32，此版本的Apache在mod_include module中有一個漏洞[30]，此module會先運算處理過被request的一些檔案後才送給client，造成攻擊者能夠利用buffer overflow改寫return address。在此實驗中，我們將Apache中已辨識出的vulnerable function之啟始指令與結束指令取代為breakpoint之後，利用Webstone

benchmark，當benchmark進行request時，若導致service執行到vulnerable function且改寫到已辨識出的CS data所在的分頁時，就會進入protection handler，而此漏洞只有在客戶端請求server-side includes file (.shtml檔案)時，才會進入vulnerable function。依照不同的server-side includes file比例(在全部請求檔案中所佔的比例)，利用Webstone benchmark測得server throughput。如Figure 10 所示，當沒有任何server-side includes file(0% .shtml)時，由於apache並沒有執行到vulnerable function，所以此時是沒有overhead的，而隨著server-side includes file的比例增高，overhead也會逐漸增高，當所有請求檔案皆為server-side includes file時會有最高的overhead(大約為 8%左右)，由此可知，儘管在所有檔案都請求都會進入vulnerable function，效能降低的比例也仍然能夠接受。因此，我們的保護機制能夠實際被使用。

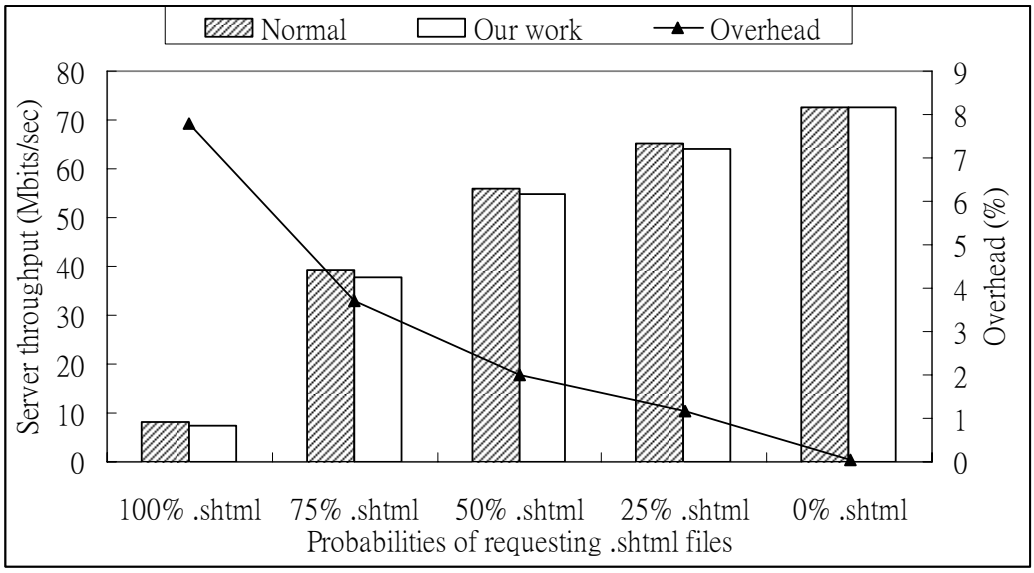


Figure 10: Performance overhead on Apache of protection

Figure 11 為我們在 server 上利用 *top* 命令測得上述 Apache 實驗的 CPU 使用率，由於在 0% .shtml 的情況下，Webstone 設定的請求檔案有部分為較大檔案，因此較 100% .shtml 的情況偏向 I/O bound，由圖中可觀察到 100% .shtml 時，會有最高的 CPU 使用率，而此時我們的保護機制，較原本 apache 的 CPU 使用率，增加了約 3.3%的使用率。而在 0% .shtml 時，則顯而易見地不會增加任何 CPU 使用量。因此能夠得知，我們的保護機制並不會造成過多的額外的 CPU 使用量。

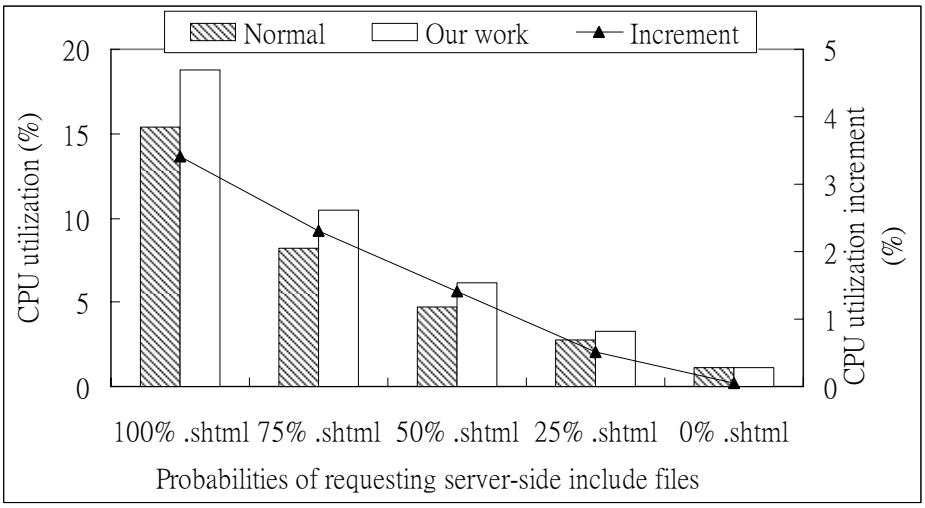


Figure 11: CPU utilization on Apache of protection

4.4.2 Sendmail

此實驗的Sendmail版本為 8.12.9，此版本的Sendmail在*prescan( )*時由於沒有檢查邊界，導致在heap data發生buffer overflow [33]。*prescan( )*是用為分析處理名稱字串並回傳一組token。在辨識出攻擊資訊之後，我們利用Postal benchmark對sendmail進行request，當進入vulnerable function時，就會因為執行了我們取代成breakpoint而進入保護處理，利用Postal測量出此版本Sendmail每分鐘接收的message數目與資料量。Figure 12 為我們的保護機制相較於正常沒有保護機制下的overhead，與我們在server上利用*top*命令測得執行Postal benchmark時的server端CPU使用量之測量結果。可發現message傳送量的overhead大約在 3%以內，而資料傳輸量的overhead約為 2%，而CPU使用率的增加量也不到 1%，因此能夠得知，在此例子中，我們的保護機制造成的效能降低，是可以被接受而能夠被實際使用的。

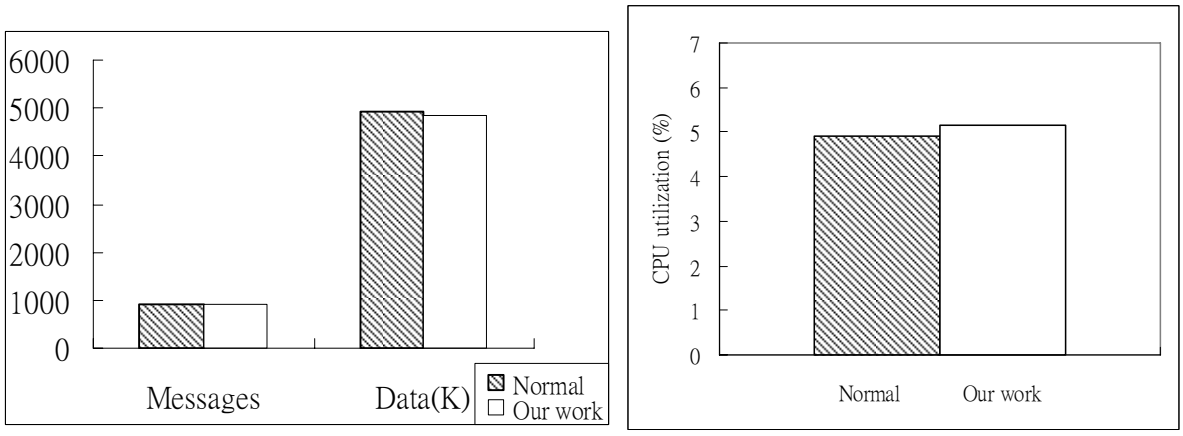


Figure 12: Performance overhead & CPU utilization on Sendmail of protection

4.4.3 Thttpd

在Thttpd version 2.22 中，當客戶端請求一個網頁伺服器中沒有的檔案(non-found file)，則Thttpd會進入*defang( )*函數以簡化客戶端輸入的字串，而此函數中的一塊buffer在被寫入時，由於沒有做好邊界檢查，因此能夠被惡意攻擊者輸入特定的字串而攻擊local data[34]。在辨識出攻擊資訊以及利用breakpoint取代vulnerable function的起始與結束指令之後，為了能夠測量抓取網頁伺服器中沒有的檔案時，server的回應時間，我們利用自己的測試程式，從第一個連線建立之前開始計時，經過中間的request與接收server回應的資料至連線結束，重複 10000 次後結束計時，以測得 10000 次request之時間。當抓取non-found file時，thttpd仍會回傳一個檔案，以告知client端使用者其請求的檔案在server中是無法找到的，因此我們仍可測得請求non-found file所需…的時間。如Figure 13，我們依照不同non-found file佔全部抓取檔案的比例測得抓取 10000 次non-found file或一個 5K大小之檔案總共需要的時間，由圖中可知，在所有檔案皆為non-found file時，我們的保護機制與原本的Thttpd皆需約六秒的時間且overhead約為 4%左右。隨著non-found file所佔的比例下降，而overhead也逐漸降低。

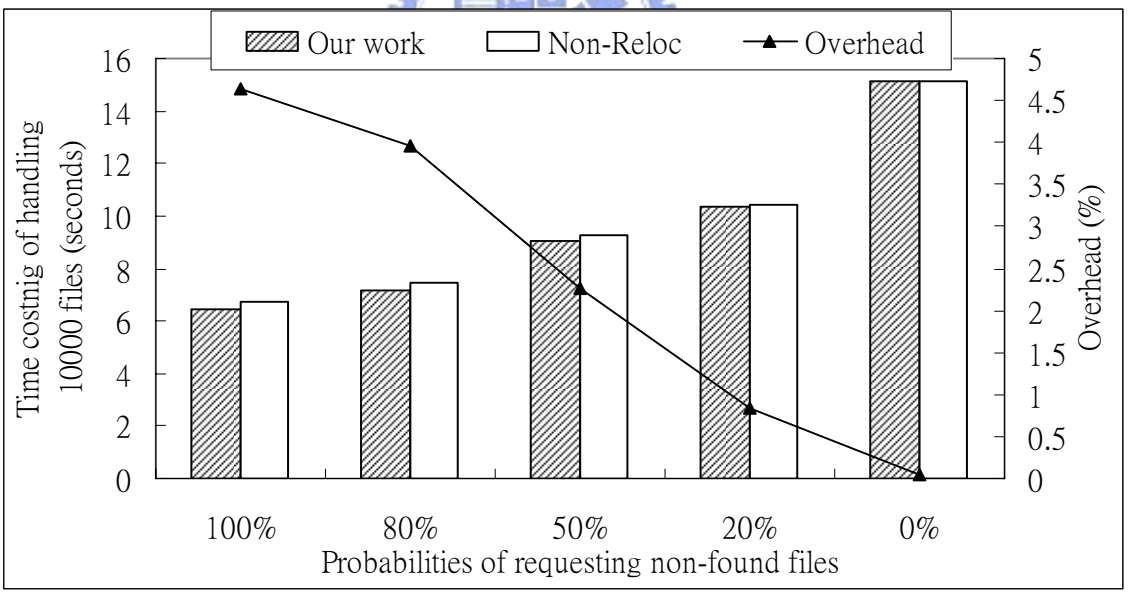


Figure 13: Performance on Thttpd of protection

4.4.4 Wu-ftpd

Wu-ftpd 2.5.0 讓攻擊者有機會在更換工作目錄(Chang Working Directory)時輸入過長的字串而改寫global data，讓攻擊者能夠控制程式至指定的位址執行程式碼[35]。在辨識出攻擊資訊之後，即vulnerable function的起始與結束指令取代成breakpoint之後，我們

利用dkftpbench benchmark測量抓取一個檔案所需的時間，為了能觀察出效能差異，我們稍微修改dkftpbench，讓每次連線都先下 40 次CWD命令之後，才抓取一次檔案，而測量的時間是從每次連線開始至檔案抓取結束，利用這樣的方式增加我們保護機制的負載，以實際觀察出效能表現差異。如Figure 14，可以發現在抓取 2 bytes大小之檔案時，會有最高的overhead，大約在 4%左右。隨著抓取之檔案越大，overhead則越來越小。這是因為在抓取較大的檔案時，所需要的時間較久，讓保護機制所產生的額外負載之比例相形之下變得較少。在抓取 5M大小之檔案時，額外負載的比例幾乎為 0。由此可知我們的機制在此情況下，是能夠被實際使用的。

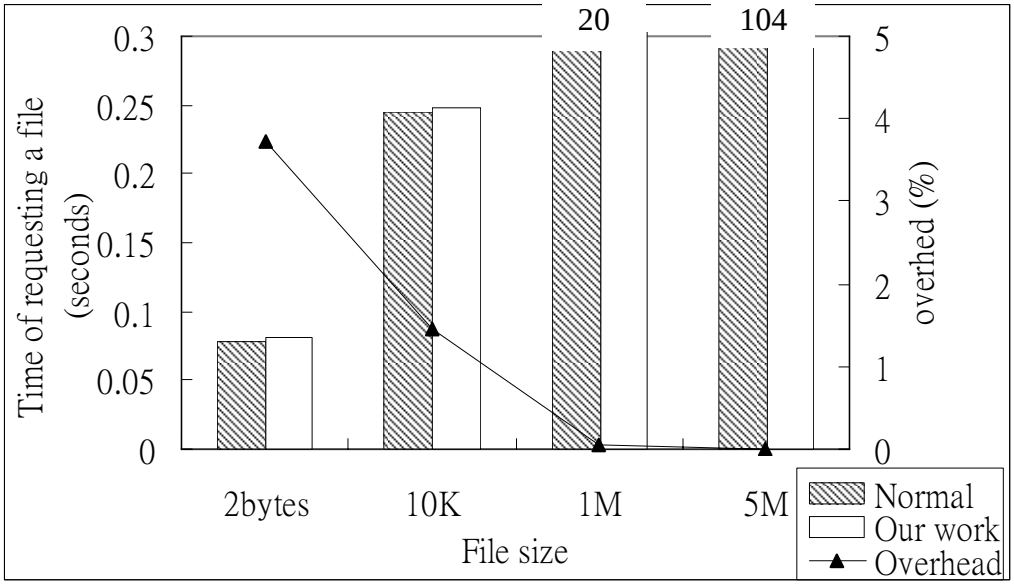


Figure 14: Performance on Wu-ftpd of protection

4.4.5 Telnetd

Telnetd version 0.17 中的 *netoprintf( )*沒有做好 *maxsize* 邊界檢查導致惡意攻擊者能利用 buffer overflow 攻擊的方式改寫 global data [43]，*netoprintf( )*會把輸入的字串參數傳送至網路上。我們利用自己的測試程式，從第一次對 server 進行 telnet 開始計時至完成第 1000 次的 telnet 連線。測量我們在辨識出攻擊資後啟動保護機制的 server，與原本無保護機制的 server 之連線時間差異。Figure 15 為利用 telnetd 測量之實驗數據，此數據為原本 Telnetd 與我們的保護機制在 telnet 1000 次所花費的時間，大約都在 50 秒左右，overhead 不到 1%，因此我們能夠得知，在此情況下保護 global data 所造成的效能降低，是能夠被接受而得以實際使用在 telnetd service 上的。

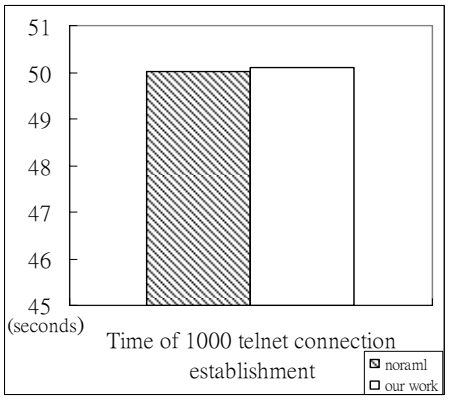


Figure 15: Performance on telnetd of protection

4.5 Performance difference between our work and non-relocated CS data

在 3.4.4 中，我們敘述了搬移 CS data 以降低保護機制之 overhead 的方式，因此接下來的實驗為測量這樣的方式所產生之效果。

4.5.1 Apache

我們利用 Apache，安排與 4.4.1 中同樣實驗環境來測量 overhead 的差異，Figure 16 中的”Non-Reloc”與”Non-Reloc overhead”，分別代表的是我們不將 CS data 搬移到分頁的起始處，如第三章所述，此時增加了進入 protection handler 的機會，而相較我們的保護機制下(圖中的”Our work”)，產生了更高的 overhead，在 100% .shtml 時約為 18%左右，較我們保護機制之 overhead 高出了 10%左右，因此可以得知我們的方式可以實際降低 overhead。

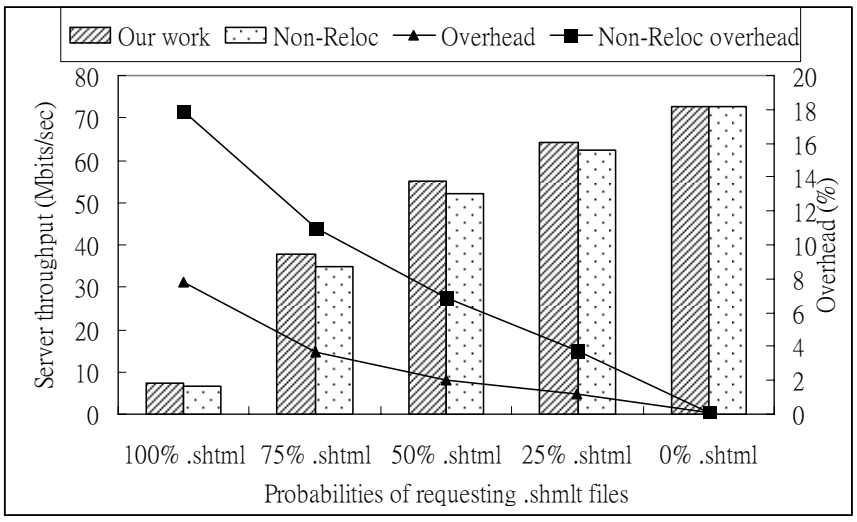


Figure 16: Performance difference on telnetd between our work and non-relocated CS data

4.5.2 Thttpd

我們亦安排與 4.4.3 中同樣實驗環境來測量 thttpd 中搬移 CS data 與否之 overhead 差異，Figure17 中的“Non-Reloc”代表沒有搬移 CS data 到分頁起始處的方式所需的時間。在 100% non-found file 時約為 9%(圖中的“Non-Reloc Overhead”)，較我們保護機制(圖中的”Our work”)的 overhead 多一倍左右。同樣的，隨著 non-found file 的比例降低，產生的 overhead 也會逐漸下降，這也是因為在伺服器找的到請求檔案的情況下，是不會進入 vulnerable function 的，所以不會有額外的負載。

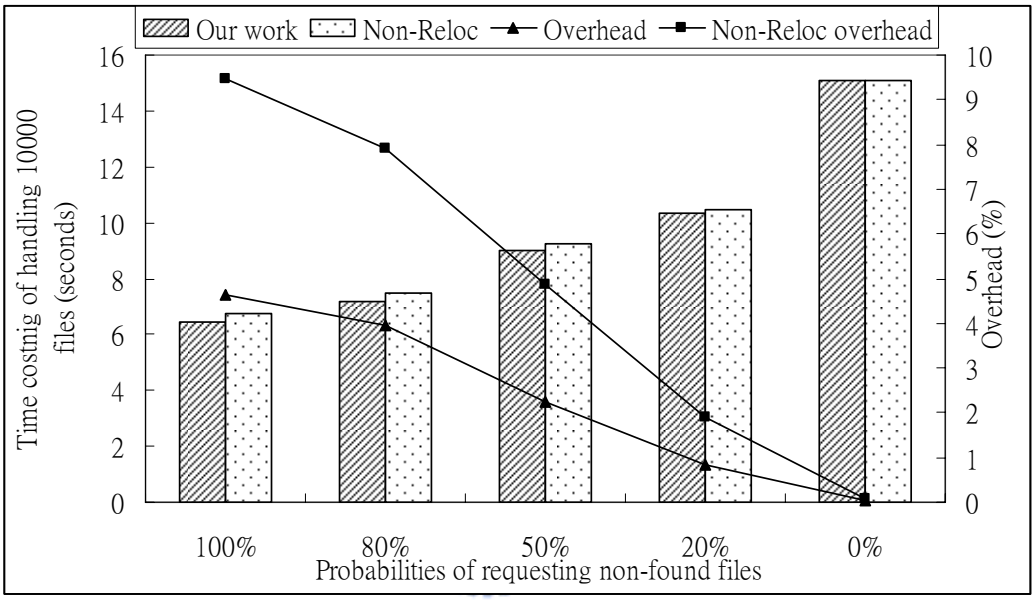


Figure 17: Performance difference on thttpd between our work and non-relocated CS data

4.5.3 Proftpd

Proftpd 1.2.7 中能夠讓攻擊者利用ascii型態傳輸時進行buffer overflow攻擊而改寫 return address[32]，我們在server執行proftpd，測量在偵測到攻擊與辨識出攻擊資訊得知 CS data為return address後，我們的保護機制與不搬移CS data的方式所產生的效能差異。利用dkftpbench benchmark測量在不同的檔案大小所需的抓取時間，可以從Figure 18 中看出，在抓取 2bytes時不搬移CS data(“non-reloc”)的方式會有 21%的overhead，相較於我們保護機制 3%的overhead要高出了許多，而隨著抓取的檔案大小增加，overhead差距也逐漸減少，這是因為抓取檔案的的時間越久，保護機制所產生的額外負載之比例相形之下會變得較少。而我們的保護機制與non-reloc最高的的差距為約為 18%左右，因此可以得知我們將CS data搬移至分頁起始，確實是能夠降低overhead的。

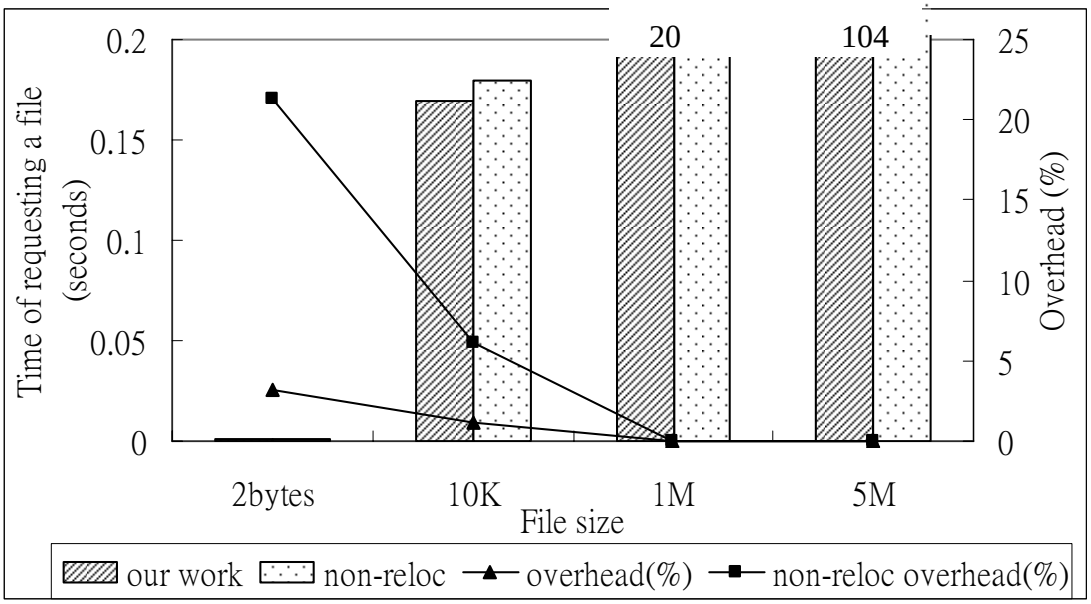


Figure 18: Performance difference on proftpd between our work and non-relocated CS data

5 Conclusion

我們提出一個自動化的系統，以針對軟體 buffer overflow 弱點部份進行保護，以有效防止攻擊者利用 buffer overflow attack 入侵控制系統。而相較於產生可能有 false alarm 的 signature，我們在偵測與辨識攻擊之後，能夠直接針對該漏洞而排除之，並自動針對該漏洞進行系統保護，並讓程式中動態配置的資料遭攻擊時也能夠辨識出以取得相關攻擊資訊，利用 ASR 以極低的 overhead 與最廣的範圍保護程式並避免其缺點－guessing attack。我們比較數種服務軟體在加上我們的保護後之效能表現差異，根據實驗結果顯示，在系統遭受攻擊前與辨識攻擊資訊時，各種服務軟體上皆只有極低的 overhead。而為了保護系統，在偵測到攻擊之後我們的保護機制在各服務軟體上亦皆只造成了不到 10%的 overhead，使我們的方案能夠被實際使用而能有效防止攻擊者利用 buffer overflow attack 入侵控制系統。

6 References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-time Defense. Against Stack Smashing Attacks. In Proceedings of the 2000 USENIX Annual Technical Conference, pages 251-262, Jun. 2000.

[2] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, A. Stefanovic, and D.D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In Proceeding of the 10th ACM Conference on Computer and Communications Security, pages 281-289, Oct. 2003.

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In Proceeding of the 12th USENIX Security Symposium, pages105-120, Aug. 2003.

[4] Carnegie Mellon University Software Engineering Institute. http://www.cert.org/

[5] C. C. Center. Cert/cc statistics 1988-2005. http://www.cert.org/stats/

[6] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In Proceedings of the 21st International Conference on Distributed Computing Systems, Apr. 2001.

[7] C. Cowan, M. Barringer, S. Beattie, and G. K.-H. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In Proceedings of the 10th Usenix Security Symposium, pages 191-199, Aug. 2001.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, pages 91-104, Aug. 2003.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Conference, pages 63-78, Jan. 1998.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In Proceedings of DARPA Information Survivability Conference and Exposition, pages 119-129, Jan. 2000.

[11] C. Dik. Non-Executable Stack for Solaris. Posted to comp.security.unix January 2, 1997.

[12] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a Realistic Tool for Statically Detecting

all Buffer Overflows in C. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 155-167, Jun. 2003.

[13] H. Etoh and K. Yoda. Protecting from Stack-Smashing Attacks. Available at http://www.trl.ibm.com/projects/security/ssp, Jun. 2000.

[14] F.-H. Hsu, F. Guo, and T.-C. Chiueh. Scalable Network-Based Buffer Overflow Attack Detection. Available at http://www.ecsl.cs.sunysb.edu/~fanglu/buffer_overflow_detection.htm

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: a Safe Dialect of C. In Proceedings of the USENIX Annual Technical Conference, pages 275-288, Jun. 2002.

[16] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In Proceedings of 3rd International Workshop on Automated Debugging, pages 13-26, May. 1997.

[17] G. S. Kc, A. D. Keromytis and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In Proceedings of the 10th ACM conference on Computer and communications security, pages 272-280, Oct. 2003.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In Proceedings of the 11th Usenix Security Symposium, pages 191-206, Aug. 2002.

[19] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 10th. USENIX Security Symposium, pages 177-190, Aug. 2001.

[20] Z. Liang and R. Sekar. Automated, Sub-Second Attack Signature Generation: A Basis for Building Self-Protecting Servers. In Proceedings of the 12th ACM Conference on Computer and Communications Security, Nov. 2005.

[21] Z. Liang and R. Sekar. Automatic Generation of Buffer Overflow Attack Signatures:An Approach Based on Program Behavior Models. In Proceedings of the 21st Annual Computer Security Applications Conference, pages 215-224, Dec. 2005.

[22] Z. Liang, R. Sekar, and D. C. DuVarney. Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems. In Proceedings of USENIX Annual Technical Conference, pages 375-378, Apr. 2005.

[23] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In Proceedings of 29th. ACM Symposium on Principles of Programming

Languages, pages 128-139, Jan. 2002.

[24] N. Nethercote and J. Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. In Proceedings of 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management, Jan. 2004.

[25] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of 12th Annual Network and Distributed System Security Symposium, Feb. 2005.

[26] The PAX team. Pax Address Space Layout Randomization. Available at http://pax.grsecurity.net.

[27] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In Proceedings of the 20th Annual Computer Security Applications Conference, pages 82-90,Dec. 2004.

[28] O. Ruwase, M. Lam. A Practical Dynamic Buffer Overflow Detector. In Proceedings of the Network and Distributed System Buffer overflow Symposium, pages 159-169, Feb. 2004.

[29] Insecure website. ADV/EXP: netkit <=0.17 in.telnetd remote buffer overflow. http://seclists.org/lists/bugtraq/2001/Aug/0108.html, Aug. 2001.

[30] SecurityFocus website. Apache mod_include Local Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/11471, Oct. 2004.

[31] SecurityFocous website. BSD fingerd buffer overflow Vulnerability. http://www.securityfocus.com/bid/2, Oct. 1988.

[32] SecurityFocus website. ProFTPD _xlate_ascii_write() Buffer Overrun Vulnerability. http://www.securityfocus.com/bid/9782, Mar. 2004

[33] SecurityFocus website. Sendmail Prescan() Variant Remote Buffer Overrun Vulnerability. http://www.securityfocus.com/bid/8641, Sep. 2003.

[34] SecurityFocus website. thttpd defang Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/8906, Oct. 2003.

[35] SecurityFocus website. Wu-ftpd message Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/726, Oct. 1999.

[36] H. Shacham, Matthew Page, B. Pfa, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, pages 298-307, Oct. 2004.

[37] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao and E. H.-M. Sha. Security Protection and Checking for Embedded System Integration against Buffer Overflow Attacks via Hardware/Software. IEEE Transactions on Computers, vol. 55, no. 4, pages 443-453, Apr. 2006.

[38] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In Proceedings of the 12th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pages 220-225, Jun. 2003.

[39] A. Smirnov and T. C. Chiueh. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In Proceedings of 12th Annual Network and Distributed System Security Symposium, Feb. 2005.

[40] Solar Designer. Non-Executable User Stack. http://www.openwall.com/linux/.

[41] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb?: The Effectiveness of Instruction Set Randomization. In Proceedings of 14th USENIX Security Symposium, Aug. 2005.

[42] StackShield. http://www.angelfire.com/sk/stackshield, Jan. 2000.

[43] D.Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Proceedings of the Network and Distributed Systems Security Symposium, pages 3-7, Feb. 2000.

[44] J. Xu, P. Ning, C. Kil, Y. Zhai and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In Proceedings of 12th ACM Conference on Computer and Communications Security, pages 223-234, Nov. 2005.