软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于应用视角的软件缓冲区溢出漏洞检测技术与工具*

司徒凌云[1,2], 李宣东[1,2], 刘杨[3]

[1](南京大学 计算机科学与技术系, 中国 南京 210023)

[2](计算机软件新技术国家重点实验室(南京大学), 中国 南京 210023)

[3](南洋理工大学 计算机科学与工程学院，新加坡 639798)

通讯作者: 李宣东, E-mail: lxd@nju.edu.cn

摘 要: 缓冲区溢出漏洞是危害最为广泛和严重的安全漏洞之一，彻底消除缓冲区溢出漏洞相当困难。学术界、工业界提出了众多缓冲区溢出漏洞检测技术与工具，面对众多的工具，使用者如何结合自身需求有效的选择工具，进而应用到漏洞的检测与修复、预防与保护、度量与评估等方面，是个具体而实际的问题。解决这一问题，需要在各异的用户需求与多样的缓冲区溢出检测技术与工具之间建立一张条理清晰、便于用户理解和使用的映射图谱。站在使用者的立场，在概述缓冲区溢出漏洞类型与特征的基础上，从软件生命周期阶段的检测与修复、缓冲区溢出攻击阶段的预防与保护、基于认识与理解途径的度量与评估三个应用视角，对缓冲区溢出漏洞检测技术与工具进行梳理，一定程度上在用户需求、检测技术与工具之间建立了一张映射图谱。

关键词: 软件安全；缓冲区溢出；漏洞检测；攻击防护；度量评估

中图法分类号: TP311

Software Buffer Overflow Vulnerability Detection Techniques and Tools based on Application Perspective

SITU Lingyun[1,2], LI Xuandong[1,2], LIU Yang[3]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[3](School of Computer Science and Engineering, Nanyang Technological University, Singapore 210023)

Abstract: Buffer overflow vulnerability is one of the most widely exploited and dangerous security vulnerabilities, eliminating buffer overflow vulnerability completely is extremely difficult. Various buffer overflow detection techniques and tools have been proposed in academy and industrial. In the face of numerous tools, it is a specific and practical issue that how could users choose these tools effectively and applied them to the application aspects such as detection and repair, prevention and protection, measurement and assessment. It is necessary to establish a clear map between various of user requirements and multiple buffer overflow detection techniques and tools for sake of solving the problem. On the basis of an overview of the types and characteristics of buffer overflow vulnerabilities, buffer overflow detection techniques ant tools are analyzed and elaborated from three application perspectives, i.e. software life cycle based detection and repair, buffer overflow attack stages based prevention and protection, knowledge and understanding based measurement and assessment, which created a map of user requirement and techniques and tools to a certain degree.

Key words: Software security; Buffer overflow; Vulnerability detection; Attack prevention and protection; Measurement and assessment

* 基金项目: 国家自然科学基金(00000000, 00000000); 南京大学计算机软件新技术国家重点实验室开放课题(KFKT00000000)

Foundation item: National Natural Science Foundation of China (00000000, 00000000); State Key Laboratory for Novel Software Technology (Nanjing University)开放课题 (KFKT00000000)

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

软件缓冲区溢出漏洞，在 CWE/SANS 排名的 25 个最危险软件漏洞中位列第三[1]，是当今危害最为广泛和严重的安全漏洞之一。缓冲区溢出指的是输入数据超过缓冲区可容纳的最大数据量，进而超出部分溢出到临近存储区域的软件漏洞。其产生的根本原因是使用非安全类型编程语言如 C/C++，强调效率优先，而对内存操作不做边界检查。缓冲区溢出可被恶意利用进而控制主机，获得系统权限，执行任意代码，对安全攸关系统造成重大危害。典型的例子最早可追溯到 1988 年爆发的 MORRIS 蠕虫[2]，其利用 BSD 操作系统后台程序的缓冲区溢出漏洞进行攻击，数天之内控制了近 6000 台网络主机，几乎导致互联网完全瘫痪，造成了近一千万美元的经济损失；再例如 2001 年 7 月爆发的 Code Red 蠕虫[3]攻击了近 36 万台服务器，造成超过 26 亿美元的损失。速度最快的属随后于 2003 年 1 月爆发的 Slammer 蠕虫[4]，其基于微软 SQL Server 的缓冲区溢出漏洞进行攻击，在 10 分钟之内感染了 7.5 万台主机，最终感染主机 60 多万，造成经济损失达 50 亿美元之巨。

时至今日，为了抵御缓冲区溢出漏洞的威胁，一方面编程人员的安全编程技能不断提升，另一方面各种缓冲区溢出漏洞的检测，防护技术也相继提出。典型的，可分为静态方法和动态方法。静态方法[5] [75][76][115]不需执行程序，基于源码分析，能够有效发现常见的缓冲区溢出漏洞。其优势在于速度快，可处理规模大，并且在一定假设前提下可以证明程序彻底摆脱某种特定类型的缓冲区溢出漏洞，其不足在于误报率和漏报率较高。相对而言，动态方法[5] [77]可获得较高的精度，代价是巨大的额外开销和性能损失。进一步的，部分动、静态结合的技术也相继提出[6][83]。上述方法在一定程度上提高了缓冲区溢出攻击的门槛，缓解了缓冲区溢出漏洞造成的危害。但是，面对当今信息社会软件规模不断扩大，软件数量不断增多（包括众多现有的 C/C++编写的系统以及以往的 C/C++遗留代码）的现实，缓冲区溢出漏洞的数目不减反增。图 1 所示的是 1989 年到 2017 年，CVE 公布的历年缓冲区溢出漏洞数目，由图 1 可知缓冲区溢出漏洞依旧是威胁软件安全的重大安全漏洞之一。

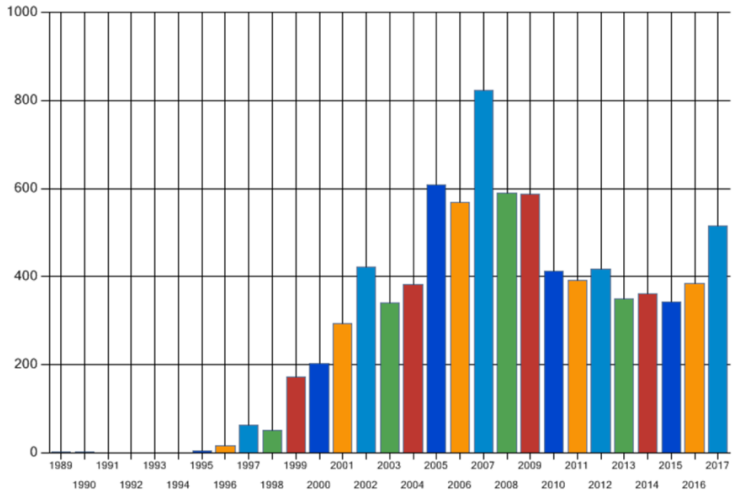

图 1 1989-2017 缓冲区溢出漏洞数目

值得一提的是，缓冲区溢出不仅仅局限于非安全类型编程语言 C/C++，安全类型的编程语言代码 Java、Perl，其底层基础同样面临的缓冲区溢出攻击的威胁[6]。

书写安全的代码可能是唯一能够使软件彻底摆脱各种安全漏洞的终极办法，软件工程领域的研究一直致力于帮助编程人员书写安全的代码。遗憾的是，依靠目前的编程范式以及各种缓冲区溢出漏洞的防护措施，要彻底的消除缓冲区溢出漏洞几乎是不可能的。因此，缓冲区溢出漏洞检测技术与工具显得至关重要，它们是检测与修复、预防与保护、度量与评估等多方面工作的基础和核心。

迄今为止，学术界和工业界提出了各种缓冲区溢出漏洞的检测技术与工具。然而，面对众多的检测技术与工具，使用者如何有效的进行选择，进而应用到缓冲区溢出漏洞的检测与修复、预防与保护、度量与评估等多个方面，是一个具体而实际的问题。

有必要对现有的软件缓冲区溢出检测技术与工具进行梳理，然而，目前对于缓冲区溢出漏洞检测技术与工具的梳理，大多基于研究者的研究视角，而非使用者的应用视角。基于研究视角，其重点关注缓冲区溢出检测方法的技术细节，以静态方法，动态方法和混合方法进行分类阐述。这样的梳理对于研究者而言清晰明了，有利于研究者深入了解各种检测技术细节进而进行技术改进。但是，对于使用者而言不够直观实用，因为使用者不关心技术与工具的进一步改进，其关心的更多的是如何在有限成本的制约下，有效的选择或者组合出能够最大限度满足自身需求的检测工具进行应用，并达到尽可能好的效果。

该命题的回答可以归结为用户需求的细分与相应检测技术工具的匹配。这需要同时深入了解用户需求和缓冲区溢出检测技术与工具。然而，一方面用户需求并不单一，是纷繁各异的，不同的行业，不同的场景下用户需求更是千差万别；同时用户需求并不独立，多个需求之间可能相互制约、需要进行多需求的平衡。另一方面，缓冲区溢出检测工具也是门类繁多，各有特色。要完成用户需求的细分与相应检测工具的匹配，在繁杂各异的用户需求与多种多样的缓冲区溢出检测技术与工具之间建立一张全面、条理清晰、而又便于用户理解、使用的映射图谱是非常困难的，同时也是非常有价值的。

本文站在使用者的立场，从技术与工具实际应用的视角出发，在概述缓冲区溢出漏洞类型与特征的基础上，从软件生命周期阶段的检测与修复、缓冲区溢出攻击阶段的预防与保护、基于认识与理解途径的度量评估三个应用视角，对缓冲区溢出缺陷检测技术与工具进行梳理，一定程度上在用户需求与检测技术与工具之间建立了一张映射图谱，为用户实际中有效选择缓冲区溢出检测技术与工具提供了指导，也为进一步的研究工作奠定了基础。

1 缓冲区溢出类型与特征

缓冲区溢出[10][79]是一种软件漏洞，对于强调效率优先的非安全类型编程语言 C/C++而言，当试图将超过缓冲区所能容纳的数据输入到缓冲区时，因其不做边界检查，就会发生溢出。其中，缓冲区指的是计算机中存储数据的一段连续内存区域,包括数据段、堆段、栈段。

当发生缓冲区溢出时，溢出的数据流入到缓冲区临近的内存区域，进而会覆盖、修改临近内存区域中的值。缓冲区溢出攻击就是利用这一特点进行的。攻击者精心构造输入内容，造成缓冲区溢出，进而使溢出部分修改附近内存中诸如返回地址，函数指针，栈帧基址，指针变量等关键类型的值，使其指向攻击者希望程序后续执行的位置，从而改变程序控制流，最终执行攻击代码（攻击代码可能位于构造的输入内容之中，也可能是系统库中的函数），实现其攻击目的。

基于上述的理解，给出缓冲区溢出漏洞和缓冲区溢出攻击的定义。

定义 1（缓冲区溢出漏洞）

缓冲区溢出漏洞是一种软件缺陷，指的是输入数据的长度超过了缓冲区能够容纳的长度，超出的部分数据溢出到临近的内存区域的一种异常。

定义 2（缓冲区溢出攻击）

缓冲区溢出攻击是攻击者基于缓冲区溢出漏洞，通过构造输入，造成缓冲区溢出，使溢出数据修改了临近内存区域的关键值（如 Return Address，Heap Metadata 等），进而劫持控制流，执行注入的或者重用已有代码（如系统库函数等）构成的攻击代码的一种软件安全攻击。

1.1 缓冲区溢出漏洞类型

缓冲区溢出漏洞按照不同的标准有不同的分类。按照缓冲区所在内存区域的位置可分为栈溢出，堆溢出和数据段溢出；按照导致溢出的内存操作函数分为字符串操作（如 strcpy 函数等）导致的溢出和格式化输出

（如 sprintf 函数等）导致的溢出等；按照溢出数据修改的关键值类型分为修改返回地址的溢出，修改函数指针的溢出，修改指针变量的溢出等。

下面简要介绍几种典型缓冲区溢出漏洞类型，主要包括栈溢出、Return-into-Libc 溢出、off-by-one 溢出、堆溢出、数据段溢出、格式化字符串溢出和整数溢出。

1.1.1 栈溢出

栈溢出是被利用最广泛的溢出漏洞。每一次函数调用，栈中会存放该函数对应的栈帧，帧中包含函数参数，函数返回地址，栈帧基址等信息。例如，函数 func 的栈帧如下图 2 所示:

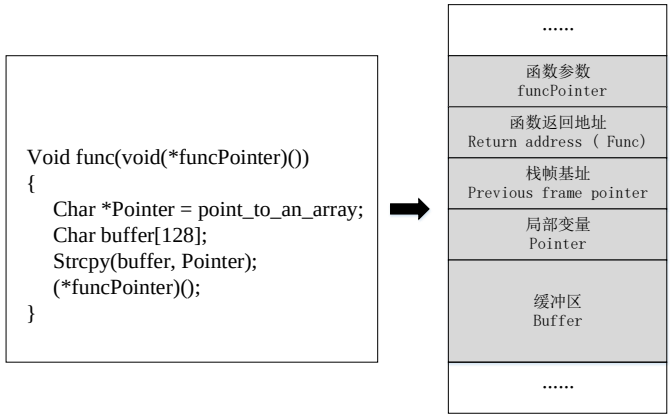

图 2 func 函数及其对应栈帧

Stack Smashing 是基于栈溢出的典型攻击，1996 年 AlephOne 在文献[7][8]中进行了详细的论述, 其基本过程如图 3 所示：首先，攻击者通过精心构造包含恶意代码的输入内容传入函数（如 Pointer 指向的数组）；然后，函数内部内存操作函数（例如 strcpy 等）将输入内容拷贝到缓冲区，进而造成溢出，溢出部分数据会修改临近缓冲区的关键值，即函数的返回地址；最后，当函数执行结束返回时，程序执行跳转到被修改过的返回地址所指向的地址，即缓冲区中恶意攻击代码所在的位置，进而执行攻击代码。

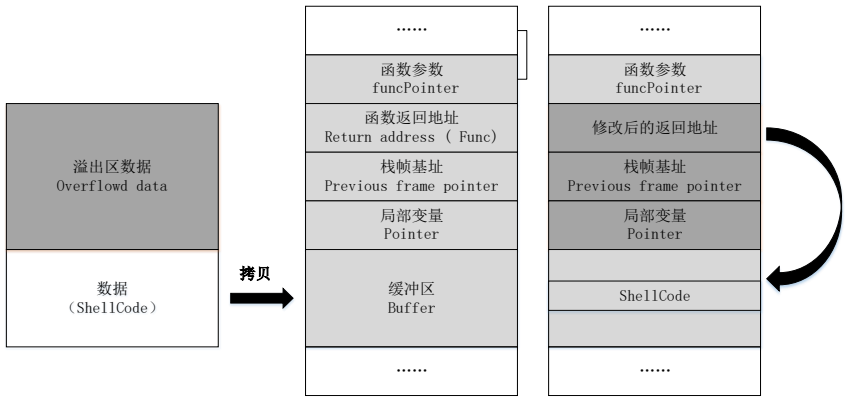

图 3 Stack Smashing 攻击示例

此外，如果攻击者不是将攻击代码注入到缓冲区中，而是重用已有代码，系统库函数（如 system( )，exec( )等）作为攻击代码，那么这样的溢出攻击叫做 Return-into-Libc 溢出攻击。ROP (Return Oriented Programming)[86][87]通过 RET 地址重用 Gadgets 代码。JOP(Jump Oriented Programming)[88][90] 通过 Call/Jmp 指令重用 Gadgets 代码构成攻击代码。函数返回地址是最重要的攻击目标之一，除此之外，指针变量、函数指针、栈帧基址等都是重要的攻击目标，即可以通过覆盖修改指针变量(如上例中的 Pointer)的值和

函数指针（FuncPointer）指向攻击代码[80]。Off-by-one 溢出则指的是输入内容恰好超出缓冲区一位数据的溢出，其通常产生于试图将一个数组中的所有元素逐个复制到缓冲区中的循环中。如下图 4 所示。

```
void notSafeCopy(char *input)
{
    char buffer [128];
    for (int i = 0; i <= 128; i ++)
        buffer [i] = input [i];
}
```

图 4 off-by-one 溢出示例

该程序意在将 input 中的数据逐个复制到长度为 128 的 buffer 数组中，但是由于 for 循环中 i<128 写成了 i<=128，所以该程序会复制 129 个数值到 buffer 中，进而造成 off-by-one 溢出.

1.1.2 堆溢出

堆是由程序运行时运用 malloc( )和 free( )等函数动态分配、释放的内存块组成，每一个内存块都包含自身内存大小和指向下一个内存块的指针等信息。虽然堆中没有函数返回地址，但是攻击者可以通过修改堆中的函数指针或者指针变量，进而达到修改程序控制流，执行攻击代码的目的。典型的，如下图 5 所示**[9]**

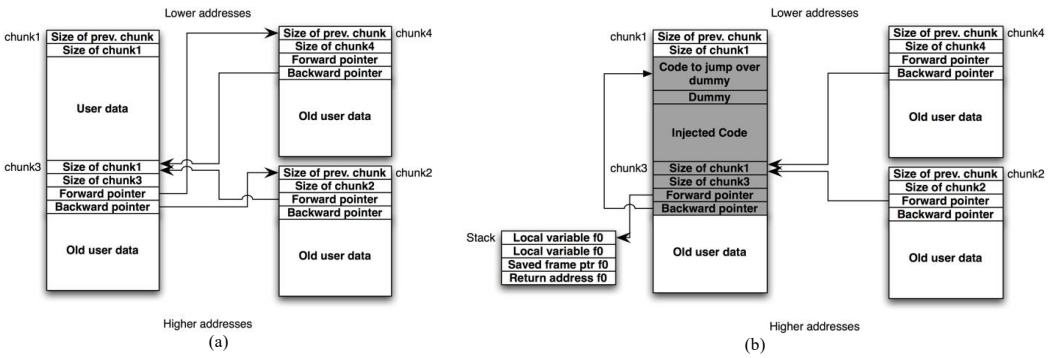

图 5 堆内存分配与堆溢出示例

图（a）展示的是一个典型的在堆中动态分配和释放的内存块情况，chunk1 是一个已分配的内存块，包含其之前存储的块的大小和它本身的大小信息，User data 部分即提供程序写入数据的 buffer 区域。chunk3 是一个临近 chunk1 且已被释放的内存块，chunk2 和 chunk4 是位于堆中其他任意位置的已被释放的内存块。chunk2,chunk3,chunk4 在一个双向链表结构中，chunk2 是链中的第一个内存块，其前向指针指向 chunk3,后向指针指向了链中前一个内存块。chunk3 的前向指针指向 chunk4,后向指针指向了 chunk2。chunk4 是链中最后一个内存块，其前向指针指向了链中下一个内存块，后向指针指向 chunk3。图(b)则展示了一个攻击，当 chunk1 中的 User data 部分溢出，攻击者将覆盖重写 chunk3 的管理信息，chunk3 的前向指针被修改指向栈中函数 f0 返回地址的前 12 个字节位置，后向指针被修改指向可以跳转到后几个字节然后执行攻击代码的代码位置（Code to jump over dummy）。当 chunk1 后续被释放，就会和 chunk3 合并成了一个大的空闲内存块。由于 Chunk3 不再是一个独立的空闲内存块，必须首先从空闲结点链表中移除 chunk3。其过程如下：chunk3->fd->bk=chunk3->bk，chunk3->bk->fd = chunk3->fd。即 fd 指向位置 12 个字节之后的内存位置的值（即 Return address f0 的地址）会被 bk 指向位置的值（即 Code to jump over dummy 地址）重写，bk 指向位置 8 字节之后的内存位置的值（dummy 内的地址）会被 fd 指向位置的值（即 Local variable f0 的地址）重写。因此在（b）中的返回地址会被一个指向跳转代码的指针重写，进而越过存储 fd 的地址区域，进而执行注入的攻击代码（Injected Code）。该技术可被用于重写任意的内存位置[102]。

1.1.3 数据段溢出

数据段溢出与堆段的溢出类似，数据段中存储的是初始化和未初始化的全局/静态变量。简单的，如下图 6 所示：

```
static char buffer [128];
static int (*fptr) (const char *str);
main (char *str)
{
        fptr = (int (*) (const char *str));
        strcpy (buffer, str);
        (void) (*fptr) (buffer);
}
```

图 6 数据段溢出示例

上述程序中，如果 str 的长度超过 buffer 容量就会造成溢出，覆盖函数指针 fptr，这样就可以改变程序的执行流程，使其跳转并执行攻击代码。

1.1.4 格式化字符串溢出

格式化字符串溢出[10][50][78]主要由格式化字符函数如 fprintf、 sprintf、 snprintf、syslog 等引起。对于格式化字符串系函数，如果不按照规定，给定正确的输入、输出格式以及相应变量，就可能发生溢出。典型的如函数 sprintf (char * str, const char * format, ...)是将格式化的数据写入 str 所指的数组中，并添加'\0'，如果格式化的数据长度超出了数组的容量就会溢出。再比如 sscanf(const char * s, const char * format, ...)从 s 中读进数据，按照 format 的格式将数据写入到其他参数中，如果格式化后得到的字符串长度大于相应的参数字符数组大小就会溢出[85]。此外，对于格式化字符串，利用%格式符可以把已经输出的字符串长度写到指定的内存单元，换言之攻击者可以通过%n 来改写函数的返回地址等信息。

1.1.5 整数溢出

整数类型规定了整型变量能存放的数值范围是固定的，如对于由 N 比特表示的无符号类型, 其表示范围是 $0\sim2^{N-1}$。当试图用一个大于或小于其取值范围的数值对其进行赋值时，或者诸如加、减、乘、左移和类型转换等操作使得其结果超出相应的范围，就会出现整数溢出缺陷。整数溢出在一般情况下不会造成安全问题，但是当溢出的整数变量用作其他类似于数组下标或者数组访问的边界控制时就造成安全问题[116]。典型的，如下图 7 所示：

```
void copy (char *str)
{
        char buffer [80];
        unsigned short len;
        len = strlen (str);
        if (len <= 80)
                strcpy (buffer, str);
        do_sth_with (buffer);
}
```

图 7 整数溢出示例

该例子中参数 str 被拷贝到一个有限长度的缓冲区中，虽然对缓冲区的写入有边界保护，但是当 str 的长度超出了短整型的范围，它很可能被截取为一个小于 80 的值，这样边界保护不再有效，一个超出缓冲区大小的数据就会被写入到缓冲区中，从而造成缓冲区溢出。

1.2 缓冲区溢出攻击特征

缓冲区溢出可发生在栈、堆、数据段区域，其主要目标是通过溢出数据修改返回地址，函数指针等关键

值，进而修改程序控制流，最终执行攻击代码。缓冲区溢出攻击的主要特征[11]可归纳为如下图 8 所示：

len:buff	len(input) > len(buffer)	输入字符串的长度大于缓冲区所能容纳的最大长度
con:addr	contains(input, type(address))	输入中包含地址类信息
con:inst	contains(input, type(instruction))	输入中包含指令信息
con:ctrl	contains(input, type(ctrlvar))	输入中包含关键变量
mod:radd	modify(retnadd)	修改返回地址
mod:fptr	modify(funcptr)	修改函数指针
mod:cvar	modify(ctrlvar)	修改关键变量
mod:cptr	modify(ctrlptr)	修改关键指针
jmp:stack	jumpe(stack)	程序执行跳转到栈内存区域
jmp:heap	jumpe(heap)	程序执行跳转到堆内存区域
exe:stack	execute(stack)	执行存储在栈中的指令
exe:heap	execute(heap)	执行存储在堆中的指令
flow:ctrl	flow(ctrlvar)	程序执行路径因为关键变量而改变

图 8 缓冲区溢出攻击特征

至此，每一类具体的缓冲区溢出攻击可以概括为上述若干特征的集合。典型的如 Stack Smashing ={len:buff, con:addr, con:inst; mod:radd,jmp:stack, exe:stack}。

2 多应用视角下的缓冲区溢出检测技术与工具

从工具实际应用的视角来看，使用者更关心的问题是如何在有限成本的制约下，有效的选择一个或多个能够最大限度满足自身需求的检测工具进行应用，并达到尽可能好的效果。解决这一问题需要进行用户需求的细分与相应检测工具的匹配，在各异的用户需求与多样的缓冲区溢出检测技术与工具之间建立一张全面、条理清晰、便于用户理解和使用的映射图谱。

虽然使用者具体的细分需求（成本、场景、精度、广度、功能等）纷繁各异，但是其使用工具的目的大致可以归纳为三点：

（1）使用工具对缓冲区溢出漏洞进行检测，进而修复软件漏洞；

（2）使用工具对缓冲区溢出攻击进行预防，进而保护软件安全；

（3）使用工具对缓冲区溢出漏洞摆脱程度进行度量，进而评估软件可信性。

基于上述归纳，本节试图从检测与修复，预防与保护，度量与评估三个应用视角出发对现有的缓冲区溢出检测技术与工具进行梳理，在用户细分需求与相应的检测技术工具之间建立一个映射图谱。

进一步，每个应用视角下的具体阐述也试图解决使用者实际中普遍关心的问题，即：

（1）在软件制品的不同阶段可分别选用哪些工具进行缓冲区溢出漏洞的检测与修复？

（2）在缓冲区溢出攻击的不同阶段可分别选用哪些工具进行缓冲区溢出攻击的预防与保护？

（3）在基于认识与理解途径的不同度量评估需求下如何选择工具度量评估缓冲区溢出漏洞摆脱程度？

以下内容的阐述一定程度上回答了上述问题，具体内容可归纳为：（1）软件生命周期阶段的检测与修复；（2）缓冲区溢出攻击阶段的预防与保护；（3）基于认识与理解途径的度量与评估。

2.1 软件生命周期阶段的检测与修复

使用者选用检测技术与工具的目的之一是为了对软件进行缓冲区溢出漏洞的检测，进而修复漏洞。然而，面对软件生命周期不同阶段产生的不同形态的软件制品，该选用哪些检测技术与工具进行分析与检测？

软件生命周期阶段可如下图 9 所示：

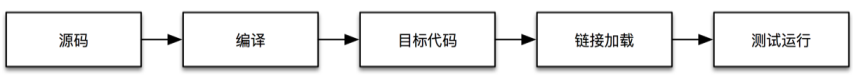

图 9 软件生命周期阶段

每一个阶段，都有相应的检测技术与工具可供用户选择进而对软件制品进行缓冲区溢出漏洞检测。在源

码阶段，用户可以使用诸如词法分析、语义分析、约束分析、符号执行、定理证明、模型检验等各种静态分析技术与工具，不需要运行程序，直接基于源码进行缓冲区溢出漏洞检测；在编译阶段，可供选择的工具主要是对现有编译器进行修改、扩展，通过增加边界信息、边界检查代码、或者增加安全类型检查来预防缓冲区溢出漏洞；在目标代码阶段，可供选择的工具更多的是基于二进制码进行缓冲区溢出漏洞分析，主流方式可概括为三种：逆向工程、补丁对比、错误注入。链接加载阶段，可供选择的工具主要是通过对不安全的库函数进行增强或替换，对系统内核或环境进行修改来阻止由不安全的库函数或者非正常的函数调用而产生安全缺陷；最后在测试运行阶段，更多的是综合使用如模糊测试、混合执行、动态污染分析等各种动态检测技术与工具，进行缓冲区溢出漏洞检测。

2.1.1 源码阶段

词法分析： 基于词法分析技术的缓冲区溢出检测工具，其基本思想是通过扫描源代码，与现有缓冲区溢出漏洞库中归纳的漏洞特征、规则等信息进行模式匹配，如果发现与漏洞库中不安全代码模式符合，则给出警告。词法分析技术，因其不考虑语义信息，所以分析速度快，可处理程序规模大，但同时误报率较高。典型的工具有 ITS4[26]、Flawfinder[27]、RATS[28]、以及商用工具 Fortify[84]等。其中 ITS4 由 Cigital 公司于 2000 年发布，其能对 C/C++程序的每一个函数进行扫描分析，与维护的缓冲区溢出漏洞库进行匹配，并依据危险等级给出提示报告，其漏洞库随着新漏洞的发现而不断更新。类似的，Flawfinder 是 2001 年发布的 C/C++安全审查工具，Flawfinder 同样内置了一个漏洞数据库，如格式化字符串溢出漏洞等。RATS 相较于前者则支持更多的语言，提供对 C、C++、Perl、PHP 以及 Python 语言的漏洞扫描。文献[29]对前三个工具性能进行了分析比较，结论指出 ITS4 的综合性能最好。类似的文献[105]对 Splint、 Fortify、 Checkmarx[110]进行了性能以及误报率，漏报率的比较分析。

语义分析： 基于语义分析的检测工具，在语法分析的基础上增加语义分析， 在发现潜在的缓冲区溢出漏洞方面具备更强的能力。代表性的工具是 Splint[29]，其为 Lint，LCLint[62] 的改进版本，其根据用户提供的语义注释信息来检测程序中的安全漏洞。系统内建立了一组使用高危函数的安全条件（如对 strcpy 要求目标函数区空间大于源缓冲区）。类似的语义分析工具还有 Prefast[69]、Prefix[70]、Marple[71]等。文献[103]构造了一组 benchmark 量化评估了如 Splint、Prefast、Clang、UNO 等典型静态分析工具的性能与准确率，结果显示 Prefast 在文中规定的测试集上对缓冲区溢出漏洞的检测准确率最高达 41.8%。

约束分析： 美国加州大学伯克利分校的 Wagner 设计的 BOON[30]是基于约束分析的缓冲区溢出检测工具的典型代表。其将字符串看成一种抽象数据类型，将缓冲区建模为一个表示大小和当前长度的整数对，对缓冲区的操作建模为对缓冲区范围的修改，进而将缓冲区的约束产生问题转化为整数范围的约束求解问题。通过对建立的程序约束条件求解来发现可能的缓冲区溢出缺陷。但是 BOON 实现的检测精度较低，实验表明 BOON 的误报率达 90%。

符号执行： 符号执行技术试图用新的方式捕获程序的执行语义，是对普通执行的一般化，其用符号代替实际输入，执行过程中，收集相关约束，到程序出口或发现错误时，根据收集的约束条件求解，进而产生测试用例。基于符号执行技术进行缓冲区溢出检测的工具有 ARMOR[86]，ARCHER[32]等，其中 ARCHER 是基于符号执行和路径敏感分析技术开发的用于分析软件中数组越界漏洞的静态分析工具。ARCHER 则通过遍历分析所有语句来发现缺陷，对于内存访问语句, 遍历模块先调用求解器检查该语句是否可能越界, 然后更新求解器的状态。基于符号执行的方法具有较高的准确性，但是由于求解器使用的开销较高，同时求解能力有限，所以基于符号执行的方法普遍存在可处理程序规模受限的问题。文献[104]构建了一组测试集量化分析了 ARCHER、Splint、UNO、BOON 等静态分析工具，结果显示 ARCHER 表现较好，检测率达 90.72%，几乎没有误报。UNO 检测率为 51.89%，同样没有误报。Splint 检测率为 56.36%，误报率为 12.03%。BOON 表现最差，检测率仅为 0.69%。

定理证明： 定理证明技术的基本思想是将源程序的行为语义和安全性质转换为逻辑公式，再将这些逻辑

公式输入到定理证明器，进而将软件缺陷的检测问题转换为逻辑公式的证明问题。如果定理证明器证明这些逻辑公式是有效的，则说明源程序不存在缺陷，其优势在于定理证明可以处理无限状态，其不足在于语句转换过程复杂，难以全面实现自动化。典型的，基于定理证明技术的缓冲区溢出检测工具有 ESC[33]和 CSSV[34]等。ESC 是可用于检测 Modula-3 和 Java 两种语言代码中的数组越界漏洞。首先输入用户注释过的程序，然后利用验证条件产生器产生关于源程序行为正确的条件，最后将产生的逻辑公式输入到定理证明器 Simplify 完成验证。如果验证失败，则说明源程序中存在错误，定理证明器给出相应的反例。CSSV 则是一种验证 C 代码中是否存在字符串溢出漏洞的工具。首先将 C 源程序和安全性质转换为带注释的程序，然后利用指针分析算法 GOLF 进行过程间别名分析，进而将源程序转换为整数程序，最后用前向完整性整数分析算法和后向完整性整数分析算法检测整数程序进而报告发现的漏洞。

模型检验： 模型检验的基本思想是使用有穷状态迁移系统对程序的行为进行建模，使用时序逻辑或者模态逻辑公式对待检验的安全性质进行刻画，程序的每条执行路径及其相应的状态对应于迁移图中的一条状态迁移轨迹。通过穷举状态迁移图中每一条状态迁移轨迹，判定该轨迹上的所有状态是否满足待检验性质，如果所有路径上的所有节点都满足性质，则程序满足了待检验性质，否则模型检验器给出使性质为假的系统状态迁移轨迹作为反例。运用模型检验方法进行缓冲区溢出检测的典型工具有 UNO[35]、MPOS[36]。其中 UNO 是一个用于发现 C 程序中数组访问越界漏洞的模型检验工具。MOPS 是 Berkeley 大学开发的可用来验证程序操作序列相关的安全性质的模型检验工具。类似的工具还有 ESPx[37]、BufSTAT[73]等。基于模型检验的方法相较于定理证明，可以实现自动化，但是只能处理有限状态程序，同时面临状态空间爆炸问题。

2.1.2 编译阶段

众多缓冲区溢出缺陷的检测工具是通过修改、扩展现有的编译器实现。主流的可分为两类：一类是通过增加边界信息以及边界检查代码。代表性的工具有 StackGuard[17]、ProPolice[18]、Purify[38]、CERD[39]、Insure++[40]、TinyCC[41]等。另一类是增加安全类型检查，代表性的工作如 CCured[42]，其通过静态分析将指针分为 SAFE、SEQ 和 WILD 三种类型，再对三类指针进行分类插装检查，如：SAFE 型的指针检查是否为空，SEQ 指针检查其地址范围等。

2.1.3 目标代码阶段

在目标代码阶段面对的往往都是二进制码，相应的基于二进制码进行缓冲区溢出检测[80][81]的工具按照使用方法可分为三类：

第一是逆向工程方法，即通过反汇编技术将二进制码转化为中间表示或者汇编代码，再基于转化后的表示进行相应的缺陷检测。典型的工具 IntScope[43], 其将可执行文件转化为中间语言表示，再运用符号执行检测潜在的整数溢出漏洞。类似的工具还有 WinSafe[72]。

第二是补丁对比方法，即通过二进制文件对比揭示二进制文件中的信息差异，进而用于缓冲区溢出漏洞挖掘。简单的，可通过比较 patch 前后可执行文件的变化，进而确定溢出点。典型的如 Todd Sabin 提出的基于指令相似性的图形化比较[45]和 Halvar Flak 提出的结构化的二进制比较[46]。前者可以发现文件中一些非结构化的变化，如缓冲区大小的改变。后者则更注重二进制可执行文件在结构上的变化。

第三是错误注入方法，即运用各种类型的不规则输入对程序进行探测，以此来触发潜在的安全漏洞，测试成功的标志是程序的异常。Fuzzing 通过向被测程序提供半有效性的输入（即可以被应用程序所接受并且具有一定破坏性的随机输入），检查应用程序是否能正确处理可能的错误输入。通过监控应用程序的执行情况发现程序中潜在的诸如缓冲区溢出等漏洞。典型的针对缓冲区溢出漏洞的检测工具有 AFL[106]、 Dowser[47]、SwordFuzzer[48]、TaintScope[49]等。Fuzzing 易实施，并能够发现确切潜在的缓冲区溢出漏洞，其主要问题是如何产生高效的测试输入尽可能覆盖目标程序的所有代码。而 Symbolic execution 能够有效的产生高覆盖度的测试用例，典型的可用于二进制的符号执行框架有 Angr[99]、S2E[100]和 BAP[101] 等。

2.1.4 链接加载阶段

一方面，链接加载的库函数中的不安全函数是导致缓冲区溢出威胁的重要推手，另一方面系统运行的软、硬件环境对缓冲区溢出攻击的实施有着重要影响。所以，在该阶段可供使用者选择用于防护缓冲区溢出漏洞的工具可分为三类：

（1）不安全的库函数增强、替换。典型的，如 FormatGuard[50]是 glibc 的一种增强，其可以在不需要显著降低程序运行性能的情况下有效检测针对格式化字符串溢出缺陷的攻击。类似的，LibSafe[51]将一些已知的易受堆栈溢出攻击和格式化字符串漏洞攻击的库函数诸如 strcpy()、printf()等进行了修改封装。这些修改后的函数一方面实现了原有功能，另一方面还可以确保缓冲区溢出范围被限制在当前栈的栈帧之内，继而返回地址和堆栈指针的内容无法被修改，可以有效的阻止堆栈溢出攻击和格式化字符串溢出攻击。LibSafeXP[52]是对 LibSafe 的拓展，而 LibVerify[53]使用和 StackGuard 类似的动态方法来进行增强保护。

（2）操作系统内核补丁实现堆栈不可执行，如 OpenBSD，PaX[23]，DEP[24]等；

（3）硬件支持不可执行内存，如 Intel AMD “NX”[25]等。

2.1.5 测试运行阶段

在测试运行阶段，基于各种动态技术诸如模糊测试、动态污染分析、混合执行方法开发的工具可供使用者选择、组合、应用于测试运行时检测缓冲区溢出漏洞。

模糊测试： 模糊测试是一种使用大量半有效输入对目标程序进行测试，通过探测、监视程序运行过程中的异常来挖掘软件系统漏洞的方法[109][113]。 模糊测试按照测试用例的产生方式可以分为基于生成的（Generation based）模糊测试与基于变异的（Mutation based）模糊测试，典型基于生成的模糊测试工具如 SPIKE[108]、Sulley[111]、Peach[112]等，通过构建协议语法与 Session 模型生成测试用例，因为其生成的测试用例满足语法格式，可以较快的通过语法检查部分，进入程序语义逻辑层面进行探索，故更适用于测试接收高结构化输入的程序；相应的，典型的基于变异的模糊测试工具如 AFL[106]、 LibFuzzer[107] 等，基于程序插桩反馈在给定 seed 基础上应用不同变异算子进而产生测试用例，通过覆盖度反馈指导不断产生可以覆盖更多路径的测试用例，更适用于测试接受输入简洁，无结构化格式要求的程序。典型基于模糊测试的检测缓冲区溢出漏洞的工具有：AFL[106]、LibFuzzer[107]、Dowser[47]、SwordFuzzer[48]、STOBO[74]、FLS[82]等。

动态污染分析：动态污点分析是在程序执行的过程中，依据执行流程标记污染数据、跟踪数据污染信息的传递，检测污染数据的非法使用，从而达到跟踪攻击路径、获取漏洞信息的目的。污点分析的过程一般可以分为三个步骤：标记污染数据，跟踪污染数据的传递，污染数据非法使用的判定。动态污点分析能有效提高了缺陷检测的精度，在阻止攻击的同时获得程序的漏洞所在，应用性较强，代表性的工具有 TaintScope[49]，Dytan[55]等。

混合执行：混合执行[117]是混合两种执行方式，在具体执行的同时对所执行到的代码进行符号执行，具体执行的特性决定了每次混合执行获取的路径都是可行路径，因此避免了误报，这是混合执行相对于静态符号执行的主要优势之一。一次混合执行结束时，其路径约束是一组约束的合取，对其中某个或某几个约束取反的同时保持其余部分不变，则可以得到路径约束的一个变体，利用求解器对变换后的新约束求解，若有解，则意味着它对应于另一条可行路径。在此过程中基于新的启发式算法探索程序的路径空间。典型的混合执行工具有：DART[54]、CUTE[56]、SAGE[58]、KLEE[57]、Angr[99]、S2E[100]和 BAP[101]等，其中 DART 和 CUTE 的路径空间搜索与路径选择策略是一次混合执行之后对取得的路径约束的最后一个约束进行取反而得到新的路径，而 SAGE 则按照不同组合对多个约束取反从而获得新的路径。KLEE 则对每条路径已执行部分进行打分，通过贪心算法从一组路径集合中选择最重要的优先处理。此外文献[113]对包括 EXE、DART、KLEE、BAP、S2E、Angr 在内的主流符号执行以及混合执行技术在内存消耗，环境交互，循环处理，状态空间搜索与路径选择，约束求解等方面进行了细致的对比分析与讨论。相应的，专门的基于混合执行检测缓冲区溢出漏

洞的工具有 BIOL[60]等。

综上所示：基于软件生命周期的检测与修复工具如下表 1 所示，一定程度上回答了用户实际关心的问题，即软件制品的不同阶段可分别选用哪些工具进行缓冲区溢出缺陷的检测与修复。

表 1 软件生命周期阶段的检测与修复工具

生命周期阶段	方法分类	工具
源码阶段	词法分析	ITS4[26]、Flawfinder[27] 、RATS[28]、Fortify[84].etc
	语义分析	LCLint[62]、Splint[29]、Prefast[69]、Prefix[70]Marple[71].etc
	整数约束	BOON[30].etc
	符号执行	ARMORY[86]、ARCHER[32].etc
	定理证明	ESC[33]、CSSV[34].etc
	模型检验	UNO[35]、MOPS[36]、ESPx[37]、BufSTAT[73].etc
编译阶段	边界检查	StackGuard[17]、Propolice[18]、Purify[38]、CERD[39].etc
	类型检查	Ccured[42].etc
目标代码阶段	逆向工程	IntScope[43]、WinSafe[72].etc
	补丁对比	Graph Isomorphism[45]，Structural comparison[46].etc
	错误注入	AFL[106]、Dowser[47]、Angr[99]、S2E[100]、BAP[101]、TaintScope[49].etc
链接加载阶段	库函数增强替换	LibSafe[51]、LibSafeXP[52]、LibVerify[53]、FormatGuard[50].etc
	系统环境修改	Pax[23]、Intel AMD “NX”[25].etc
测试运行阶段	模糊测试	AFL[106]、LibFuzzer[107]、Dowser[47]、SwordFuzzer[48]、STOBO[74].etc
	动态污染分析	TaintScope[49]、Dytan[55].etc
	混合执行	DART[54]、CUTE[56]、KLEE[57]、SAGE[58]、EXE[59]、BOIL[60].etc

2.2 缓冲区溢出攻击阶段的预防与保护

使用者使用检测技术与工具第二个目的是为了预防缓冲区溢出攻击，进而保护软件产品。然而，问题是面对不同特征的缓冲区溢出攻击，用户可以从哪些阶段进行预防，不同阶段又可以选用哪些工具达到预防与保护的目的？

典型的缓冲区溢出攻击阶段如下图 10 所示：

图 10 缓冲区溢出攻击阶段

每一个攻击阶段，用户都可以选用相应的技术与工具构建预防和保护措施。典型的措施有：输入检测，边界检查，关键值完整性检查，控制流监控和堆栈不可执行。

2.2.1 输入数据阶段

正如文献[12]中所言：“所有的输入都是恶意的，除非被证明”，外部的恶意输入是所有攻击产生的源头，OSWAP 更是于 2017 年将“不安全的攻击保护（包括不恰当的输入验证）”列为 Top 10 安全风险之一[90]。故在输入数据阶段，相应的预防保护是对输入数据进行检查。具体技术与工具可分为两类：一类是基于攻击代码特征的模式匹配，就缓冲区溢出攻击而言，如果其恶意代码是在输入数据中，那么输入数据必然包含一些典型特征（如：包含与关键值类型相同的值，用于填充的 NOP 指令，Shellcode 等），因此运用模式匹配算法将输入数据与已知的恶意攻击代码特征进行匹配，如果发现恶意输入，给出警告，就可以一定程度的在源头有效控制恶意攻击的发生，典型的工具如 Polygraph [13]；另一类是标记并跟踪分析输入的污染数据，其指导思想是恶意输入的污染数据如果在后续敏感操作中被使用，就有可能造成威胁。所以通过标记输入，对其进行污染传播分析，如果污染数据在后续的敏感操作中被使用，就给出警告，典型的工具有 Tainted Pointer [14]。

2.2.2 缓冲区溢出阶段

缓冲区溢出是缓冲区溢出攻击发生的前提，最直接的对应于缓冲区溢出阶段的保护策略是增加边界检查。

所谓边界检查指的是对每一个数据的操作进行检查，使其必须在缓冲区的边界之内进行。

缓冲区溢出检测工具按照边界检查实现的方式可分为三类：

(1) 基于编译器的修改增加相应的边界信息和边界检查代码，典型的工具如 Bounds Checker[15]，CERD[39]，MOBD[61]等;

(2) 基于虚拟机的修改增加安全类型检查，典型的工具如 CCured[42];

(3) 通过硬件实现支持，如 IA-32/ I432[16]。

理论上，如果对每一个内存操作都进行边界检查，就可以彻底避免缓冲区溢出攻击的发生。但是需要付出巨大的额外开销，如果对 C/C++的每一个内存操作都进行边界检查，也就损失了 C/C++效率优先的优势。

2.2.3 修改关键值阶段

缓冲区溢出的直接目的是运用溢出数据覆盖、修改诸如返回地址、函数指针等关键类型的值。那么，在关键值使用之前，对其进行完整性检查有助于发现缓冲区溢出漏洞，预防缓冲区溢出攻击。

典型的，以返回地址为例，对其进行完整性检查的方法有三种：

(1) 基于 Canary 检测的保护，即在返回地址前增加一个 Canary 字节数据，将对返回地址的完整性检查转化为对 Canary 值的完整性检查。因为如果缓冲区溢出数据覆盖修改了返回地址，必然覆盖修改 Canary，故在使用返回地址之前对 Canary 进行检查，如果发现 Canary 值已经被修改，则中止程序给出警告。典型的 StackGuard [17]对编译器 GCC 添加补丁，使得在函数入口能自动在栈中生成 Canary 标记，在函数调用结束时检测 Canary 标记是否改变来发现并且阻止缓冲区溢出攻击。ProPolice[18]与 StackGuard 类似，不同的是 ProPolice 将函数中用到的局部变量重新排列，使得函数的缓冲区紧邻 Canary，栈中的其他关键值（如函数指针等）就不会受到缓冲区溢出的影响。

(2) 基于加密的保护，即将加密技术运用于保护关键值（返回地址等）的完整性。用预先定义的密钥在返回地址存储到内存前进行加密，使用时再进行解密。典型的如 PointGuard[19], 其通过改进 GCC 编译器实现，基本思想是在程序装入内存时，先将指针型数据加密，当程序访问这些数据时，再在寄存器中进行解密。

(3) 基于备份的保护，即将返回地址复制一份备份，存储在其他区域，当使用时通过与备份比对或者用备份替换来保证返回地址的完整性。典型的工具如 StackGhost[20]、StackShield[21]、RAD[44]等。 StackShield 基于对 GCC 的修改，在函数调用将返回地址压栈时，用一个全局数组存储备份返回地址。当函数调用结束时，用备份的返回地址替换栈中的返回地址，以此达到保护返回地址的目的。StackGhost 思想类似，具体实施是基于 SPARC 处理器体系结构，对系统内核进行修改，从而实现对返回地址的保护。

基于关键值完整性检查的保护策略可以有效的对特定类型的关键值进行保护，从而提高了相应缓冲区溢出攻击的门槛。其不足在于因为是针对特定类型的关键值，所以其只能检测特定类型的缓冲区溢出漏洞。

2.2.4 改变控制流阶段

攻击者修改关键值的目地是为了改变程序控制流，程序执行进行非正常的跳转或者非正常的系统调用，是出现缓冲区溢出攻击的重要特征。因此，保障程序控制流的完整性[88]，通过阻止非正常的程序执行跳转和非正常的系统调用，使程序运行中的控制转移，始终处于原有的控制流图限定范围内，可以有效的预防和阻止缓冲区溢出攻击。

控制流完整性（CFI）[93]的具体做法是通过分析程序的控制流图，获取间接转移指令（包括间接跳转，简洁调用以及函数返回指令）目标的白名单，并在运行过程中核对间接转移指令的目标是否在白名单中。CFI 从实现角度可分为细粒度和粗粒度两种，细粒度 CFI 严格控制每一个间接转移指令的转移目标，这种精细的检查提升了安全性，但是通常会引入巨大开销。粗粒度 CFI 则是将一组类似或相近类型的目标归到一起检查，以降低开销，但会导致安全性的下降[94][95]。基于 CFI 保护的典型工作包括 CCFIR[94], 其策略是区分间接调用指令和函数返回指令，阻止未经验证的返回指令跳转到铭感操作函数上，一定程度上避免了 CFI 插桩开

销大的问题。BinCFInally[95]则提出致力于将 CFI 应用于已有的商用应用上。PathArmor[96]则是一个可用于现实应用程序的高效可靠的上下文敏感 CFI 方案。此外 KCoFI 将 CFI 做到了操作系统内核之中，使之免受经典的控制流劫持，return2user 和代码段修改攻击[97]。文献[98]则对主流的控制流完整性保护技术做了详细的性能分析，比较与讨论。 此外，保障控制流完整性的工具有基于 API-Hook 方法的 AIFD[22]，其通过跟踪返回地址相关的 API 调用，与收集的合法 API 调用模式进行比较而发现潜在的缓冲区溢出漏洞。McAfee[89]有着类似的缓冲区溢出保护机制，例如若堆栈中的 shellcode 调用了 getProcaddress 等函数，那么 McAfee 就会中止当前进程并报警。

2.2.5 执行攻击代码阶段

缓冲区溢出攻击的最终目的是执行攻击代码，如果最终使得攻击代码无法执行，那么就可以在最后阶段成功阻止攻击，避免危害。基于该指导思想的保护策略是使堆栈不可执行来阻止攻击代码存储在堆栈中的缓冲区溢出攻击。

典型的实现方式有两种，其一是通过操作系统内核补丁实现，典型的工具如 PaX[23]，DEP[24]；其二是通过处理器体系结构支持实现，如 SPARC，AMD “NX”[25]等。

不可执行内存的保护方法可以有效的阻止攻击代码存储在堆栈中的攻击，但是对于攻击代码不在堆栈中的攻击，如 Return-into-Libc、ROP、JOP 攻击就无效。针对 Return-into-Libc、ROP、JOP 攻击，一般配合使用堆栈不可执行如 DEP[24]和 随机化地址空间 ASLR（Address Space Layout Randomization）机制[90]，可以起到很好的防护效果。

综上所示，基于缓冲区溢出攻击阶段的预防与保护工具如下表 2 所示，一定程度上回答了用户实际关心的问题，即在缓冲区溢出攻击的不同阶段可分别选用哪些工具进行缓冲区溢出攻击的预防与保护。

表 2：缓冲区溢出攻击阶段的预防与保护工具

攻击阶段	方法分类	工具
输入数据阶段（输入检测）	基于攻击代码特征检测	Polygraph[13].etc
	基于污染数据分析	Taint Pointer[14].etc
缓冲区溢出阶段（边界检查）	基于编译器修改	Bounds Checker[15]、CERD[39]、MOBD[61].etc
	基于虚拟机修改	CCured[42].etc
	基于硬件修改	IA-32/I432[21].etc
修改关键值阶段（完整性检查）	基于 Canary	StackGuard[17]、ProPolice[18].etc
	基于加密	PointGuard[19].etc
	基于备份	StackGhost[20]、StackShield[21]、RAD[44].etc
改变控制流阶段（控制流监控）	基于 CFI 的保护	PathArmor[96]，KCoFI[97]**错误!未找到引用源。**.etc
	基于 API-Hook	AIFD[22].etc
	基于非正常的系统函数调用	McAfee[89].etc
执行攻击代码阶段（不可执行内存）	基于系统内核修改	PaX[23]、DEP[24].etc
	基于硬件结构支持	AMD “NX”[25]、ADSL[90].etc

2.3 基于认识与理解途径的度量与评估

使用者使用检测技术与工具的第三个目的是对软件摆脱缓冲区溢出漏洞的程度进行度量，进而评估软件安全性。 如下图 15 所示，对软件摆脱缓冲区溢出漏洞的程度进行度量与评估是通过基于用户需求建立度量规约，选择与用户需求相适应的缓冲区溢出漏洞检测工具对软件进行分析，并基于分析所得的对软件形成的直接的认识与理解，包括检测出的漏洞数目，各项度量指标值，度量规约的完备度等基础上进行的。度量与评估的对象是软件系统，但同时也体现着对缓冲区溢出漏洞检测工具能力的度量。

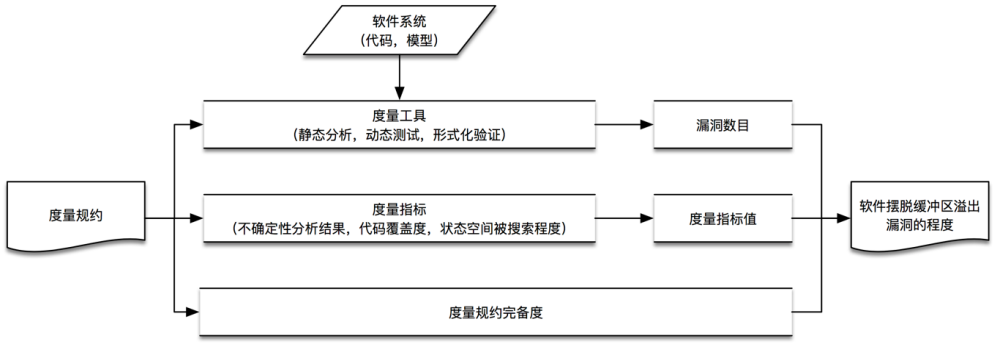

图 11 软件缓冲区溢出漏洞摆脱程度度量与评估

度量是定量的，指的是基于工具获得客观证据；评估是定性的，指的是基于度量结果（即获得的客观证据）形成主观判断。直接认识与理解的维度可分为：对模型认识与理解的程度；对代码认识与理解的程度。直接认识与理解的途径可分为：静态分析，动态测试，形式化验证。

用户根据需求选择度量工具进行缓冲区溢出漏洞摆脱程度度量与评估的基本流程可如下图 12 所示：

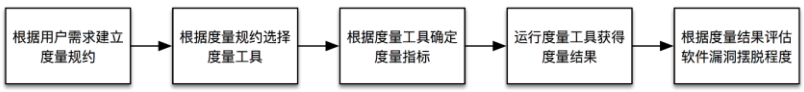

图 12 软件缓冲区溢出漏洞摆脱程度度量与评估流程

首先，根据用户需求建立度量规约。度量规约是由一组对应于各个途径下检测工具的评估项构成，其根据用户需求建立，体现用户需求和可投入成本，同时也是度量结果可信性比较的依据。度量规约本身根据其完备级别有着可信级别的划分。实际中，需要充分针对具体应用场景，对多种需求进行平衡，进而建立相应的度量规约。通常的，对缓冲区溢出漏洞而言，用户选用相应工具应用到漏洞摆脱程度度量与评估时的度量规约一般涉及为以下几个方面：

（1）成本：指的是用户可投入的经济成本。表现为具体选择时需要考虑工具是免费开源还是商用付费，对于商用付费类型，试用、购买、租用等不同方式的价格等因素。

（2）效率：指的是用户针对特定类型或规模的软件系统进行度量时可允许的时间限制。表现为具体选择时需要一方面关注工具检测算法的复杂度，另一方面关注工具处理规模与耗时的比值。

（3）广度：指的是需要检测哪些类型的缓冲区溢出漏洞。表现为具体选择时关注于工具所能检测的缓冲区溢出漏洞的类型以及运行平台。

（4）精度：指的是用户需要对工具检测软件产品缓冲区溢出漏洞的准确程度的要求，表现为选用工具是更多的关注指误报率与漏报率。

（5）功能：指的是用户考虑检测工具功能模块的要求，具体选择是关心的可能是工具是否同时包括检测和对应的修复功能，从而可以降低人工确认修复的负荷和成本等。

（6）途径: 指的用户对技术手段和认识与理解途径的要求。表现为选择度量工具时关注于工具使用的技术途径，如静态分析，动态测试，形式化验证或者多种途径结合。

（7）其他: 如用户需要对软件制品生命周期的哪些阶段进行检测，或者预防攻击的那些阶段进行检测保护等。

然后，根据度量规约选择度量工具。度量工具检测的项目内容对应于度量规约中的评估项。为此，度量规约同时决定了度量工具选择的标准，如度量规约中“广度”规定了选择的度量工具能够检测的缓冲区溢出漏洞类型，“途径”需求规定了选择的度量工具基于的认识与理解途径，包括静态分析途径，动态测试途径，形

式化验证途径。

（1）静态分析途径

静态分析途径指的是基于静态分析工具对软件代码制品进行扫描，进而发现代码中的语法、语义错误。一般采用驼鸟策略，基于分析结果形成警告信息。其优点是可直接检查程序，而不用驱动程序运行，但是其警告信息需要进一步确认，成本较高。静态分析途径下主要的分析技术包括有词法分析、语义分析、约束分析、符号执行等，静态分析更多的在软件测试运行前的阶段使用。

（2）动态测试途径

动态测试途径指的是动态运行程序以发现错误。其优点在于检测可以达到较高的精度，但是性能和规模受限。动态测试途径下主要的分析技术包括模糊测试、污染分析、混合执行等。

（3）形式化验证途径

形式化验证途径指的是通过建立系统的形式化模型，进而验证模型是否满足给定性质。但是其构建的数学模型状态空间复杂度较高，此外状态空间爆炸是其面临的主要问题。形式化验证途径下主要的分析技术包括定理证明和模型检验等。

下表 3，4 分别给出了缓冲区溢出漏洞类型以及不同认识与理解途径下对应的缓冲区溢出漏洞检测工具。

表 3：缓冲区溢出漏洞类型与对应缓冲区溢出漏洞检测工具

缓冲区溢出漏洞类型	专用类型检测工具	通用类型检测工具
栈溢出	StackGuard[17]、StackShield[21]、PointGuard[19]、ProPolice[18]、RAD[44]、StackGhost[20]、SafeStack[63].etc	ITS4[26] Flawfinder[27] RATS [28] Fortify [84] Splint [29] Bounds Checker[15] Dowser [47] Purify[38]
堆溢出	ContraProlice [68]、Valgrind[91].etc	
数据段溢出	ValueGuard[64].etc	
格式化字符串溢出	FormatGuard[50]、cqual[85].etc	
整数溢出	IntScope[43] 、 SwordFuzzer[48] 、 RICB[65] 、 RICH[66] 、 IntPatch[67] .etc	

表 4：认识与理解途径下的缓冲区溢出漏洞程度漏洞检测工具

认识与理解途径	方法分类	度量工具
静态分析途径	词法分析	ITS4[26]、Flawfinder[27]、RATS[28]、Fortify[84].etc
	语义分析	LCLint[62]、Splint[29]、Prefast[69]、Prefix[70]、Marple[71].etc
	整数约束	BOON[30].etc
	符号执行	ARMORY[86]、ARCHER[32].etc
动态测试途径	模糊测试	AFL[106]、Dowser[47]、SwordFuzzer[48]、STOBO[74]、FLS[82].etc
	动态污染分析	TaintScope[49]、Dytan[55].etc
	混合执行	CUTE[56]、KLEE[57]、SAGE[58]、EXE[59] 、BOIL[60].etc
形式化验证途径	定理证明	ESC[33]、CSSV[34].etc
	模型检验	UNO[35]、MOPS[36]、ESPx[37]、BufSTAT[73].etc

然后，根据度量工具确定度量指标。其中，度量指标是相关于分析、测试和验证技术与工具的，其值反映认识与理解（即：分析、测试、验证）的充分度。例如，静态分析中的相关指标可以是不确定性分析结果的比例；动态测试的相关指标可以是各种模型覆盖度、代码覆盖度；形式化验证的相关指标可以是状态空间被搜索的程度。接着，运行度量工具获得度量结果，度量结果指的是运行度量工具对软件制品进行检测获得的对应于度量规约的客观证据，如检测的漏洞数目，对应的度量指标值。

最后，根据度量结果评估软件缓冲区溢出漏洞摆脱程度。评估是主观判断，基于度量结果,在综合考虑度量规约的完备度、检测出的漏洞，度量指标反映的认识与理解的充分度的基础上，结合不同领域、行业、机构、项目以及个人经验设定度量规约中个度量项的可信阈值，在不彻底消除漏洞的情况下给出判定。

综上所示，一定程度上回答了用户实际关心的问题，即在如何基于认识与理解途径，在不同需求下选择缓缓冲区溢出漏洞检测工具获得对软件的认识与理解，进而度量与评估软件缓冲区溢出漏洞摆脱程度。

3 总结与展望

站在使用者的立场，在概述缓冲区溢出漏洞类型与特征的基础上，从软件生命周期阶段的检测与修复、缓冲区溢出攻击阶段的预防与保护、基于认识与理解途径的度量与评估三个应用视角，对缓冲区溢出漏洞检测技术与工具进行梳理，一定程度上在用户需求与检测技术工具之间建立一张映射图谱，为用户实际中有效选择缓冲区溢出检测技术与工具提供了指导，为进一步的研究工作奠定了基础。

目前，缓冲区溢出漏洞检测技术与工具的研究中，存在一些后续研究工作可以关注的方面:

（1）误报率与漏报率是衡量单个缓冲区溢出检测工具能力的重要指标，也是对比多个类似检测工具能力的有效依据。然而目前相关文献中工具能力的判定更多的是基于个例的测试样本，看能否发现未知的漏洞，很少给出具体的误报率与漏报率的数值。其根源在于衡量工具自身检测能力，或者对比多个工具检测能力时，缺乏一个统一的完备的标准测试集（Benchmark）作为权威的测试对象，以便产生令人信服的误报率与漏报率的数值报告。

（2）就单一工具缓冲区溢出检测方法与技术改进而言，静态方法处理规模大，速度快，但是误报率与漏报率高；动态方法检测结果精确，但是额外开销与性能损失大。动、静态方法的结合是进一步改进检测效率与精度的有效方式。目前，混合执行近十年来得到了学术界大量关注，在遍历程序执行路径，探索程序执行状态空间方面有着突出优势。其次，模型检验方法通过建模，性质规约，通过穷举模型状态空间判定性质是否满足，在一定程度上可以证明系统彻底摆脱某种类型的缓冲区溢出漏洞。结合模型检验与混合执行的方法，对系统动态语义进行精确建模，提炼缓冲区溢出漏洞性质进行规约，运用类似于混合执行的方法探索状态空间，可以进一步降低缓冲区溢出检测的误报率与漏报率。

（3）误报率的产生受两方面因素影响：其一是检测技术对程序动态语义建模的准确程度，其二是对缓冲区溢出漏洞特征提炼的精确程度。漏报率的产生更多的是由对程序执行状态空间探索的充分度决定。然而，目前大部分检测工具都是基于某一个关键特征(如缓冲区数据溢出，关键值被修改等)，在某一个特定阶段（源码阶段，运行时阶段等），关注于单一工具的检测技术的改进提高。缺乏基于对多用户需求与应用场景要求深入分析、理解、平衡基础上进行工具匹配组合用于多特征，多阶段检测，检测结果融合比对分析，进而发挥工具组合效益的研究。

所以，结合上述的三点可以关注改进的地方，未来的研究内容如下：

（1）缓冲区溢出漏洞最小完备标准测试集收集整理。收集、整理一个用于测试缓冲区溢出检测工具自身检测能力，用于对比测试类似工具检测能力的最小完备标准测试集（Benchmark）。基于该权威测试集，测试工具可以运用该测试机进行自测，给出准确的误报率与漏报率信息。

（2）基于模型检验与混合执行的检测技术改进，降低误报率、漏报率。检测方面结合模型检验与混合执行的方法，对程序动态语义进行尽可能的精确建模，基于对特定类型（如格式化字符串溢出，或整数溢出）漏洞特征深入提炼基础上进行性质规约，借鉴混合执行思路，将缓冲区溢出检测问题转化为状态可达性求解问题。降低误报率与漏报率。

（3）基于应用场景的多用户需求平衡与多工具组合应用研究。针对具体应用场景要求，对多用户需求进行深入细致分析、理解、平衡，选择与需求相匹配的工具组合应用，进行多阶段、多特征的缓冲区溢出缺陷检测，进而发挥工具组合效益。基于应用场景的多用户需求平衡与多工具组合应用进行检测，对多工具检测结果进行对比、分析、确认、分级。所有的特征都满足肯定就是漏洞，满足一部分的（其中满足关键特征的，可能性就大，一般特征的就可能性就小）可能是漏洞，都不满足肯定不是漏洞。

References:

[1] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/

[2] Spafford E H. The Internet worm program: An analysis [J]. ACM SIGCOMM Computer Communication Review, 1989, 19(1): 17-57.

[3] Moore D, Shannon C. Code-Red: a case study on the spread and victims of an Internet worm[C]//Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement. ACM, 2002: 273-284.

[4] Moore D, Paxson V, Savage S, et al. Inside the slammer worm [J]. IEEE Security & Privacy, 2003, 1(4): 33-39

[5] Li P, Cui B. A comparative study on software vulnerability static analysis techniques and tools[C]//Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on. IEEE, 2010: 521-524.

[6] Piromsopa K, Enbody R J. Survey of Protections from Buffer-Overflow Attacks [J]. Engineering Journal, 2011, 15(2): 31-52.

[7] One A. Smashing the stack for fun and profit [J]. Phrack magazine, 1996, 7(49): 14-16.

[8] Piromsopa K, Enbody R J. Buffer-overflow protection: the theory[C]//Electro/information Technology, 2006 IEEE International Conference on. IEEE, 2006: 454-458.

[9] Younan Y, Joosen W, Piessens F. Efficient protection against heap-based buffer overflows without resorting to magic [M]//Information and Communications Security. Springer Berlin Heidelberg, 2006: 379-398.

[10] Lhee K S, Chapin S J. Buffer overflow and format string overflow vulnerabilities [J]. Software: Practice and Experience, 2003, 33(5): 423-460.

[11] Bishop M, Engle S, Howard D, et al. A taxonomy of buffer overflow characteristics[J]. Dependable and Secure Computing, IEEE Transactions on, 2012, 9(3): 305-317.

[12] Howard M and Leblanc D. "Chapter 10: all input is evil! " in Writing Source Code, 2nd ed: Microsoft Press, 1965.

[13] Newsome J, Karp B, Song D. Polygraph: Automatically generating signatures for polymorphic worms[C]//Security and Privacy, 2005 IEEE Symposium on. IEEE, 2005: 226-241.

[14] Chen S, Xu J, Nakka N, et al. Defeating memory corruption attacks via pointer taintedness detection[C]//Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on. IEEE, 2005: 378-387.

[15] Cowan C, Wagle P, Pu C, et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade[C]//DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings. IEEE, 2000, 2: 119-129.

[16] Gehringer E F, Keedy J L. Tagged architecture: how compelling are its advantages? [C]//ACM SIGARCH Computer Architecture News. IEEE Computer Society Press, 1985, 13(3): 162-170.

[17] Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks[C]//Usenix Security. 1998, 98: 63-78.

[18] Etoh H, Yoda K. ProPolice: GCC extension for protecting applications from stack-smashing attacks[J]. IBM (April 2003), http://www.trlibm. com/projects/security/ssp, 2003.

[19] Cowan C, Beattie S, Johansen J, et al. Pointguard TM: protecting pointers from buffer overflow vulnerabilities[C]//Proceedings of the 12th conference on USENIX Security Symposium. 2003, 12: 91-104.

[20] Frantzen M, Shuey M. StackGhost: Hardware Facilitated Stack Protection[C]//USENIX Security Symposium. 2001, 112.

[21] Shao Z, Zhuge Q, He Y, et al. Defending embedded systems against buffer overflow via hardware/software[C]//Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003: 352-361.

[22] Han H, Lu X L, Ren L Y, et al. AIFD: a runtime solution to buffer overflow attack[C]//Machine Learning and Cybernetics, 2007 International Conference on. IEEE, 2007, 6: 3189-3194.

[23] Van der Veen V, Cavallaro L, Bos H. Memory errors: the past, the present, and the future[M]//Research in Attacks, Intrusions, and Defenses. Springer Berlin Heidelberg, 2012: 86-106.

[24] Microsoft: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003 (September 2006)

[25] Krazit T. PCWorld-News-AMD Chips Guard Against Trojan Horses[J]. IDG News Service, 2004.

[26] Viega J, Bloch J T, Kohno Y, et al. ITS4: A static vulnerability scanner for C and C++ code[C]//Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference. IEEE, 2000: 257-267.

[27] Flawfinder. Available: http://www.dwheeler.com/flawfinder/

[28] RATS. Available: http://www.securesw.com/rats/

[29] Wilander J, Kamkar M. A comparison of publicly available tools for static intrusion prevention[J]. 2002.

[30] Evans D, Larochelle D. Improving security using extensible lightweight static analysis[J]. software, IEEE, 2002, 19(1): 42-51.

[31] Wagner D, Foster J S, Brewer E A, et al. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities[C]//NDSS. 2000: 2000-02.

[32] Xie Y, Chou A, Engler D. Archer: using symbolic, path-sensitive analysis to detect memory access errors[C]//ACM SIGSOFT Software Engineering Notes. ACM, 2003, 28(5): 327-336.

[33] Detlefs D L, Leino K R M, Nelson G, et al. Extended static checking[J]. 1998.

[34] Dor N, Rodeh M, Sagiv M. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C[C]//ACM Sigplan Notices. ACM, 2003, 38(5): 155-167.

[35] Holzmann G J. Static source code checking for user-defined properties[C]//Proc. IDPT. 2002, 2.

[36] Chen H, Wagner D. MOPS: an infrastructure for examining security properties of software[C]//Proceedings of the 9th ACM conference on Computer and communications security. ACM, 2002: 235-244.

[37] Hackett B, Das M, Wang D, et al. Modular checking for buffer overflows in the large[C]//Proceedings of the 28th international conference on Software engineering. ACM, 2006: 232-241.

[38] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors[C]//In Proc. of the Winter 1992 USENIX Conference. 1991.

[39] Ruwase O, Lam M S. A Practical Dynamic Buffer Overflow Detector[C]//NDSS. 2004.

[40] Zhivich M, Leek T, Lippmann R. Dynamic buffer overflow detection[C]//Workshop on the evaluation of software defect detection tools. 2005.

[41] Poletto M, Hsieh W C, Engler D R, et al. C and tcc: a language and compiler for dynamic code generation[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1999, 21(2): 324-369.

[42] Necula G C, Condit J, Harren M, et al. CCured: type-safe retrofitting of legacy software[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005, 27(3): 477-526.

[43] Wang T, Wei T, Lin Z, et al. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution[C]//NDSS. 2009.

[44] Chiueh T, Hsu F H. RAD: A compile-time solution to buffer overflow attacks[C]//Distributed Computing Systems, 2001. 21st International Conference on. IEEE, 2001: 409-417.

[45] Sabin T. Comparing binaries with graph isomorphisms[J]. Bindview. http://www. bindview. com/Support/RAZOR/Papers, 2004.

[46] Flake H. Structural comparison of executable objects[J]. 2004.

[47] Haller I, Slowinska A, Neugschwandtner M, et al. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations[C]//Usenix Security. 2013: 49-64.

[48] Cai J, Zou P, He J, et al. A Smart Fuzzing Approach for Integer Overflow Detection[J]. INFORMATION TECHNOLOGY IN INDUSTRY, 2014, 2(3): 98-103.

[49] Wang T, Wei T, Gu G, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection[C]//Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010: 497-512.

[50] Cowan C, Barringer M, Beattie S, et al. FormatGuard: Automatic Protection From printf Format String Vulnerabilities[C]//USENIX Security Symposium. 2001, 91.

[51] Baratloo A, Singh N, Tsai T. Libsafe: Protecting critical elements of stacks[J]. White Paper http://www. research. avayalabs. com/project/libsafe, 1999.

[52] Lin Z, Mao B, Xie L. LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions[C]//Information Assurance Workshop, 2006 IEEE. IEEE, 2006: 332-339.

[53] Baratloo A, Singh N, Tsai T K. Transparent Run-Time Defense Against Stack-Smashing Attacks[C]//USENIX Annual Technical Conference, General Track. 2000: 251-262.

[54] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[C]//ACM Sigplan Notices. ACM, 2005, 40(6): 213-223.

[55] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C]//Proceedings of the 2007 international symposium on Software testing and analysis. ACM, 2007: 196-206.

[56] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C[M]. ACM, 2005.

[57] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]//OSDI. 2008, 8: 209-224.

[58] Molnar D, Li X C, Wagner D. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs[C]//USENIX Security Symposium. 2009, 9.

[59] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death[J]. ACM Transactions on Information and System Security (TISSEC), 2008, 12(2): 10.

[60] Rawat S, Mounier L. Finding buffer overflow inducing loops in binary executables[C]//Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. IEEE, 2012: 177-186.

[61] Kuang C, Wang C, Huang M. Memory-Size-Assisted Buffer Overflow Detection[J]. Journal of Software, 2014, 9(2): 336-342.

[62] Evans D, Guttag J, Horning J, et al. LCLint: A tool for using specifications to check code[J]. ACM SIGSOFT Software Engineering Notes, 1994, 19(5): 87-96.

[63] Chen G, Jin H, Zou D, et al. Safestack: automatically patching stack-based buffer overflow vulnerabilities[J]. Dependable and Secure Computing, IEEE Transactions on, 2013, 10(6): 368-379.

[64] Van Acker S, Nikiforakis N, Philippaerts P, et al. ValueGuard: Protection of native applications against data-only buffer overflows[M]//Information Systems Security. Springer Berlin Heidelberg, 2010: 156-170.

[65] Wang Y, Gu D, Xu J, et al. RICB: Integer overflow vulnerability dynamic analysis via buffer overflow[M]//Forensics in Telecommunications, Information, and Multimedia. Springer Berlin Heidelberg, 2011: 99-109.

[66] Brumley D, Chiueh T, Johnson R, et al. RICH: Automatically protecting against integer-based vulnerabilities[J]. Department of Electrical and Computing Engineering, 2007: 28.

[67] Zhang C, Wang T, Wei T, et al. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time[M]//Computer Security–ESORICS 2010. Springer Berlin Heidelberg, 2010: 71-86.

[68] Krennmair A. ContraPolice: a libc extension for protecting applications from heap-smashing attacks[J]. 2003.

[69] Microsoft Prefast. http://www.microsoft.com/whdc/devtools/tools/prefast.mspx.

[70] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors[J]. Software-Practice and Experience, 2000, 30(7): 775-802.

[71] Le W, Soffa M L. Marple: a demand-driven path-sensitive buffer overflow detector[C]//Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008: 272-282.

[72] Park S H, Han Y J, Hong S J, et al. The Dynamic Buffer Overflow Detection and Prevent ion Tool for Yindows Executables Using Binary Rewr iting[C]//Advanced Communication Technology, The 9th International Conference on. IEEE, 2007, 3: 1776-1781.

[73] Radosavac S, Seamon K, Baras J S. Short Paper: bufSTAT-a tool for early detection and classification of buffer overflow attacks[C]//Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on. IEEE, 2005: 231-233.

[74] Haugh E, Bishop M. Testing C Programs for Buffer Overflow Vulnerabilities[C]//NDSS. 2003.

[75] Shahriar H, Zulkernine M. Classification of static analysis-based buffer overflow detectors[C]//Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on. IEEE, 2010: 94-101.

[76] Padmanabhuni B, Tan H. Techniques for defending from buffer overflow vulnerability security exploits[J]. 2011.

[77] Cowan C, Wagle P, Pu C, et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade[C]//DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings. IEEE, 2000, 2: 119-129.

[78] Shankar U, Talwar K, Foster J S, et al. Detecting Format String Vulnerabilities with Type Qualifiers[C]//USENIX Security Symposium. 2001: 201-220.

[79] Khedker U P. Buffer Overflow Analysis for C[J]. arXiv preprint arXiv:1412.5400, 2014.

[80] Alouneh S, Bsoul H, Kharbutli M. Protecting Binary Files from Stack-Based Buffer Overflow[M]//Information Science and Applications. Springer Berlin Heidelberg, 2015: 415-422.

[81] Chen C M, Chen S M, Ting W C, et al. An enhancement of return address stack for security[J]. Computer Standards & Interfaces, 2015, 38: 17-24.

[82] Shahriar H, Zulkernine M. A Fuzzy Logic-Based Buffer Overflow Vulnerability Auditor[C]//Dependable, Autonomic and Secure Wang Q, Wu SJ, Li MS. Software defect prediction.

[83] Wang W, Lei Y, Liu D, et al. A combinatorial approach to detecting buffer overflow vulnerabilities[C]//Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on. IEEE, 2011: 269-278.

[84] Fortify http://www.fortify.net/

[85] Shahriar H, Zulkernine M. Mutation-based testing of format string bugs[C]//High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE. IEEE, 2008: 229-238.

[86] Chen L H, Hsu F H, Hwang Y, et al. ARMORY: An automatic security testing tool for buffer overflow defect detection[J]. Computers & Electrical Engineering, 2013, 39(7): 2233-2242.

[87] Kornau T. Return oriented programming for the ARM architecture[D]. Master's thesis, Ruhr-Universität Bochum, 2010.

[88] Bletsch T, Jiang X, Freeh V W, et al. Jump-oriented programming: a new class of code-reuse attack[C]//Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011: 30-40.

[89] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity[C]//Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005: 340-353.

[90] Snow K Z, Monrose F, Davi L, et al. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization[C]//Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013: 574-588.

[91] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[92] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]//ACM Sigplan notices. ACM, 2007, 42(6): 89-100.

[93] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity[C]//Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005: 340-353.

[94] Zhang C, Wei T, Chen Z, et al. Practical control flow integrity and randomization for binary executables[C]//Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013: 559-573.

[95] Zhang M, Sekar R. Control Flow Integrity for COTS Binaries[C]//USENIX Security Symposium. 2013: 337-352.

[96] Yan der Veen V, Andriesse D, Göktaş E, et al. Practical context-sensitive CFI[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015: 927-940.

[97] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels[C]//Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014: 292-307.

[98] Burow N, Carr S A, Nash J, et al. Control-flow integrity: Precision, security, and performance[J]. ACM Computing Surveys (CSUR),

2017, 50(1): 16.

[99] http://angr.io/

[100] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. ACM SIGPLAN Notices, 2011, 46(3): 265-278.

[101] Brumley D, Jager I, Avgerinos T, et al. BAP: A binary analysis platform[C]//International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, 2011: 463-469.

[102] Kaempf M. Vudo-an object superstitiously believed to embody magical powers[J]. 2001.

[103] Cifuentes C, Hoermann C, Keynes N, et al. BegBunch: Benchmarking for C bug detection tools[C]//Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). ACM, 2009: 16-20.

[104] Kratkiewicz K J. Evaluating static analysis tools for detecting buffer overflows in c code[R]. HARVARD UNIV CAMBRIDGE MA, 2005.

[105] Ye T, Zhang L, Wang L, et al. An empirical study on detecting and fixing buffer overflow bugs[C]//Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on. IEEE, 2016: 91-101.

[106] http://lcamtuf.coredump.cx/afl/

[107] https://llvm.org/docs/LibFuzzer.html

[108] Biyani A, Sharma G, Aghav J, et al. Extension of SPIKE for encrypted protocol fuzzing[C]//Multimedia Information Networking and Security (MINES), 2011 Third International Conference on. IEEE, 2011: 343-347.

[109] http://fuzzing.org/

[110] https://www.checkmarx.com/

[111] http://www.fuzzing.org/wp-content/SulleyManual.pdf

[112] https://www.peach.tech/

[113] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery[M]. Pearson Education, 2007.

[114] Baldoni R, Coppa E, D'Elia D C, et al. A survey of symbolic execution techniques[J]. arXiv preprint arXiv:1610.00502, 2016.

附中文参考文献:

[115] 夏一民,缓冲区溢出漏洞的静态检测方法研究[D],2007.

[116] 孙浩, 曾庆凯. 整数漏洞研究: 安全模型, 检测方法和实例[J]. Journal of Software, 2015, 26(2).

[117] 李舟军, 张俊贤, 廖湘科, 等. 软件安全漏洞检测技术[J]. 计算机学报, 2015, 38(4): 717-732.