基于内存更新记录的漏洞攻击错误定位方法

葛 毅 茅 兵 谢 立
(南京大学计算机科学与技术系 软件新技术国家重点实验室 南京 210093)

摘 要 软件漏洞攻击威胁日益严重。其中基于内存腐败漏洞的攻击最为普遍,如缓冲区溢出和格式化串漏洞。提出一种针对内存腐败漏洞攻击的自动错误定位方法。基于内存更新操作记录,可以回溯找到程序源代码中腐败关键数据的语句,从而提供有益的信息修复漏洞并生成最终补丁。

关键词 内存腐败攻击,软件安全,错误定位

Automatic Fault Localization to Memory Corruption Vulnerabilities Based on Memory Update Log

GE Yi MAO Bing XIE Li
(Department of Computer Science and Technology, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract Attacks exploiting vulnerabilities in software are becoming a great threat to the society. The most common attack method is to exploit memory corruption vulnerabilities such as buffer overflow and format string bugs. This paper presented a fault localization approach to automatically identify both known and unknown memory corruption vulnerabilities. Based on memory update log, we can trace back to the statement in source code that is tricked to corrupt critical data. The proposed techniques can provide useful information in fixing the vulnerabilities and generating the real patch.

Keywords Memory corruption attack, Software security, Fault Localization

1 引言

随着计算机的广泛应用和互联网的迅速发展,计算机软件越来越庞大和复杂。软件的安全问题已成为计算机系统安全的最大问题所在[1]。所谓软件的安全问题指的是由于软件的研究、设计、开发、测试及使用过程中的缺陷而造成的安全漏洞(security flaws)。典型的例子包括缓冲区溢出(buffer overflow)漏洞[2]、格式化串(format string)漏洞[3]、malloc/free错误[4]等。针对这些程序漏洞的攻击层出不穷。

近十多年来,为应对程序漏洞攻击的巨大威胁,人们提出了很多自动化程序漏洞攻击防范方法。到目前为止,已经有很多的方法从不同角度给出了防范程序漏洞攻击的不同解决方案。比如以StackGuard[5]对返回地址区域附加的canary标记是否改变来检测溢出是否发生;以RAD(Return Address Defender)[6]对返回地址保存到其他区域等措施来允许溢出但不让程序改变执行流等;通过使堆栈不可执行(如PaX[7])以及入侵检测手段等使得即使改变了程序执行流但不让攻击代码执行。这些防范方法的重点主要集中在攻击检测方面,提供的供后续处理的信息有限,使得通常的后续处理做法只是最简化的处理,如结束进程、重启服务等,并没有从根本上解决安全问题。虽然很多攻击防范方法可以阻止部分漏洞攻击,平缓攻击效果和规范化程序行为,但是人们仍然需要知道程序漏洞的位置,更新程序补丁,从根本上修复程序漏洞。

软件漏洞修复的最终途径是官方补丁。然而在对2005年11月至2006年7月之间发现的微软Windows操作系统的10个漏洞的研究发现,在漏洞被发现之后,微软公司需要平均75天才能发布漏洞补丁包[8]。当软件漏洞被发现之后,通常是由软件开发人员通过手工的代码审计、跟踪、调试等手段进行漏洞定位并发布漏洞补丁。然而,由于当前计算机程序日益复杂,这种手工的修补方式导致程序漏洞的修补时间明显地落后于程序漏洞攻击时间。因此,为了及时发布程序补丁,需要快速定位软件漏洞。

尽管在软件生命周期开发调试阶段提出了很多一般性的错误定位方法来找到导致程序错误的原因并定位程序中相关代码,包括程序切片[9,10],Delta调试[11]等,但这些方法多是针对程序开发调试,其定位目标、定位粒度、定位效率等都不适用于针对漏洞攻击的错误定位。所谓漏洞攻击错误定位是指程序发生漏洞攻击后能找到导致此次攻击的根源。这些错误定位方法缺乏对漏洞攻击的认识,考虑过多漏洞攻击错误定位无须考虑的细节,导致定位结果粗糙、效率低下、自动化程度低。

为了克服传统错误定位方法在满足针对漏洞攻击错误定位的局限性,近年来的部分研究工作已经开始结合攻击检测技术来辅助漏洞攻击的错误定位。比如XunJun等在文献

到稿日期:2008-02-28 本文的研究工作得到国家863高技术研究计划(No. 2007AA01Z448),国家自然科学基金(60773171)和江苏省自然科学基金(BK2007136)的资助。

葛 毅 硕士,主要研究方向为软件安全,E-mail:geyi614@gmail.com;茅 兵 教授,博导,主要研究方向为系统安全与分布式系统;谢 立 教授,博导,主要研究方向为分布式系统。

[16]提出了一种针对程序漏洞的定位方法，其定位目标是直接导致内存腐败的指令，但由于缺乏足够的信息，导致过多依赖具体的指令系统和调试工具，系统复杂，需要反复执行程序来定位具体指令。而且在很多情况下，定位的指令并不是本地代码空间的指令，而是系统空间的库函数指令，因此并不能定位源程序本身调用点的漏洞根源。

本文提出一种基于内存更新记录的漏洞攻击错误定位方法：利用程序分析技术获得软件源程序中漏洞攻击的可能点，由此插入代码来设置记录动态内存更新操作，运行时根据攻击检测技术检测漏洞攻击，由动态记录的内存更新日志，准确回溯定位到源程序级别的漏洞所在。后续工作不管是自动生成虚拟补丁还是辅助生成官方补丁都有所裨益。

本文第 2 节首先介绍基于内存更新记录的漏洞攻击错误定位方法，接着在第 3 节介绍系统的设计与实现，第 4 节为相关工作的比较，最后是小结。

2 基于内存更新记录的漏洞攻击错误定位

我们注意到缓冲区溢出、整数溢出、格式化串等大多数内存腐败(memory corruption)攻击通常遵循一个模式，称之为控制数据攻击(control-data attack)或控制流攻击[20]：它们改变目标程序的控制数据(如返回地址和函数指针等)，执行插入的恶意代码或上下文无关的库函数代码(如 return-into-library 攻击)。这些攻击的共性是利用程序漏洞恶意篡改某些内存地址的内容，目的是植入恶意代码和改变控制流相关的内存变量。

由于内存腐败攻击的必要条件之一是利用程序内部某些内存指令，恶意篡改某些内存变量，因此我们提出专门针对内存腐败漏洞攻击的错误定位方法。我们的目标是找到漏洞攻击的源程序级别的根源所在，即源程序中哪条语句恶意篡改了某内存地址存放的关键控制流对象数据，直接导致攻击的发生。

下面是一个缓冲区溢出攻击的典型例子。

```
1 #include(string. h)
2 int main(int argc,char * * argv)
3 {
4       char buf[40];
5       strcpy(buf,argv[1]);
…       …
n        return0;
n+1   }
```

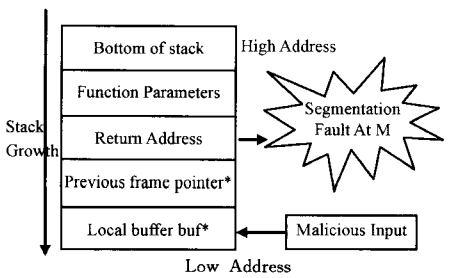

图 1 覆盖 Return Address 攻击场景

攻击者可以在图 1 中所示的函数的局部缓冲区(buf)中植入攻击代码，并且通过 strcpy()这种不进行边界检查的函数来覆盖程序的返回地址(Return Address)或者前一个栈帧指针(previous frame pointer)。当然，这时需要攻击者精心地构造适当的覆盖地址，攻击才能成功。这种情况还包括覆盖分配在堆栈中的函数指针的攻击。在此例中，程序漏洞根源是第五行的 strcpy 语句，没有做边界检查导致缓冲区溢出的发生，攻击者用恶意输入 M 覆盖了返回地址(Return Address)所在的内存地址；而在第 n 行程序 return 返回语句引用被恶意篡改的返回地址(M)时导致程序错误(Segmentation Fault at M)。本文错误定位的目标是定位到真正的漏洞攻击点 strcpy 语句，即漏洞根源，而不是直接导致程序出错的 return 语句。

首先我们需要检测漏洞攻击，在捕获到攻击之后进行错误定位。已有的漏洞攻击检测技术，如地址随机化[7]，都是在错误发生点才捕获攻击。由于出错点与攻击点在时间和空间上的跨度，我们缺乏足够的信息从错误发生点回溯定位到攻击点。因此我们希望程序在运行时能动态记录广义的审计信息，为后续的错误定位提供依据。我们结合静态源码分析技术，通过给程序添加额外的程序语义，动态记录内存更新操作，从而通过内存更新日志回溯找到漏洞攻击发生点，直接定位到源程序语句。出于性能代价考虑，并不是所有内存操作都需要记录，我们通过静态源程序分析建立简单的软件安全模型，找到可能的漏洞(攻击点)，通过源程序的插入改写，在这些可能的攻击点动态记录内存更新操作。

在捕获到内存腐败漏洞攻击之后，自动诊断进程内存镜像及寄存器内容，找到攻击目标的内存地址，从而查找内存更新日志，回溯定位到恶意篡改该内存地址的语句。此例中导致内存异常错误的线性地址为 M(Segmentation Fault at M)，其内存地址为栈上返回地址(Return Address)所在位置 location(M)。我们在内存更新日志中查找最后一条用内容 M 篡改了内存地址 location(M)的语句，即为我们需要定位的漏洞攻击点。

3 系统设计与实现

为了验证方法的有效性，我们在 Linux 上实现了基本的原型系统。

图 2 是我们系统的整体框架。

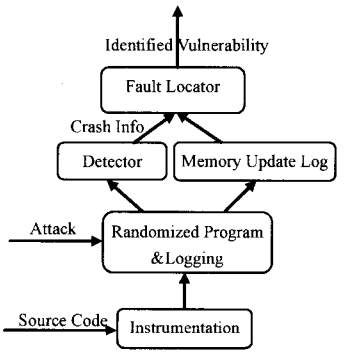

图 2 系统框架图

我们的实现工作分 3 个部分：第 1 部分是程序分析和程序转换，使用 CIL[17] 工具在源程序插入一些代码，用以运行时动态记录内存更新操作；第 2 部分是攻击检测器(Detector)，利用 PaX[7] 地址随机化技术捕获漏洞攻击；第 3 部分是错误定位(Fault Locator)，我们实现了内核补丁来读取寄存

器并分析内存镜像，根据内存更新日志回溯定位漏洞攻击点。下面介绍各个方面的工作。

3.1 程序转换(Instrumentation)

程序转换的目的是让程序能在运行时动态记录内存更新操作。出于性能代价的考虑，我们通过静态源码分析得到程序中可能的漏洞所在，记录在这些可能的攻击点所进行的内存更新操作，而不是记录所有的内存更新操作。正确的程序语义引起的内存操作并不会影响程序的正常执行。我们观测到大部分漏洞攻击都利用了不安全的库函数操作，如格式化串攻击利用了 vprintf 函数中部分代码片段；缓冲区溢出则利用 strcpy，memcpy 等不检测边界的库函数；malloc/free 错误利用了 malloc 库函数的漏洞。因此，我们实现的原型系统暂时只记录调用这些不安全的库函数引起的内存更新操作。实际的数据也表明，我们收集的存在漏洞的 Gnu 工具集，其程序漏洞都是由于调用这些不安全的库函数。

我们使用 CIL[17] 工具扫描程序源代码，识别出程序代码中需要记录的库函数调用。在这些库函数调用之前，CIL 插入一小段代码，使程序能在运行时动态记录库函数调用产生的内存更新操作。CIL 是一种高层次的中间语言表示以及配套的工具集，可以很方便地用于程序分析和程序转换。

插入的代码使得运行时动态生成记录项，记录下此次库函数调用的源程序文件名、调用行号以及产生的内存更新操作。内存更新日志的每个日志项包括 5 项数据：file_name，line_number，write_address，len，data。其中 write_address 为此次内存更新操作的起始地址，data 为具体内存操作的内容。

表 1 是我们需要记录的库函数。在这些可能的攻击点记录发生的内存更新操作。

表 1 需要记录的库函数

Function class	Libc functions
Copying	Memcpy(), strcpy(), strncat(),
Format string	Sprintf(), snprintf(), vprintf(), vsnprintf(),
Memory management	Malloc(), calloc(), realloc(), free(), strdup()
Network I/O	Readv(), recv(), recvfrom()
File I/O	Read(), fread(), scanf(), vscanf(), fscanf(), vfscanf(), gets(), fgets()

3.2 攻击检测(Detector)

攻击检测负责捕捉漏洞攻击并触发自动错误定位例程。我们可以利用大部分已有的针对内存腐败攻击的检测技术，例如 StackGuard、地址随机化技术、指令随机化技术等。在本文中我们选择地址随机化技术，其优点是性能代价小，可以检测大部分的基于内存腐败的攻击。

随机化方法其实就是带类似加密特征的程序运行方式。此类方法的出发点是，无论哪种类型的攻击，其一个重要的前提条件就是对软件在运行时候的内存组织或者运行方式存在一定的先验知识，通过对这些软件特征进行随机化(类似加密)，软件的运行方式可以相对难以受到软件漏洞攻击的影响。典型的工作如 PaX[7]，利用 ASLR(Address Space Layout Randomization)技术在运行时随机化进程的地址空间分布，使得攻击者很难合理推断其攻击目标、恶意代码和系统库函数的内存分布位置，达到抵御攻击的目的。攻击者的恶意输入将导致系统产生内存异常错误，我们截获该异常信号，触发错误定位例程。

3.3 错误根源定位 (Fault Locator)

利用检测器(Detector)监视被保护程序的执行过程，一旦攻击发生，我们捕捉到内存异常信号，自动触发错误定位。通过注册我们的信号处理函数(SIGBUS，SIGSEGV，SIGILL 信号)，分析此时进程内存镜像，查找内存更新日志回溯定位到漏洞根源。Linux 的信号机制提供了访问进程上下文的方法，例如可以得到在定位分析过程中需要的进程相关寄存器的内容。我们实现了内核补丁提供查询寄存器内容的功能。首先我们需要得到导致内存异常的线性地址 M，在漏洞攻击的场景下，M 为攻击者精心设计的恶意输入。在 IA32 平台架构上，M 存放在 CR2 寄存器。接着我们需要分析 M 存放的地址 location(M)。根据被腐败内存的种类，分 3 种情况：返回地址的篡改、函数指针的篡改和数据指针的篡改。第一种情况，根据 x86 架构上函数返回指令的语义，帧栈指针寄存器(SP)存放着 location(M)+1。我们根据导致内存异常的线性地址 M 和当前指令指针 IP 一致，确定这是一次返回地址腐败的攻击。这种攻击，我们向内核查询，获取 SP 寄存器内容，对应 location(M)=SP−1。后两种情况，通用的办法需要精确的二进制反汇编和数据流分析，目前还没有高效的方法。基于原型实现的简易和实用性考虑，我们参照 COVERS[18] 中的方法：首先，基于大部分攻击是针对帧栈的攻击，这种情况属于返回地址的篡改，不需要复杂的二进制分析。其次，大部分其他类型的攻击基于一些有漏洞的库函数调用上，例如堆溢出攻击是利用堆内存管理函数中代码片段(malloc)；格式化串(format-string)攻击基于 vprintf 部分代码片段。因此我们静态构造一个这些漏洞代码的序列表，根据这些代码片段中的指令序列回溯找到 location(M)[18]。最后，在内存更新日志里回溯查找最后一条向内存地址 location(M)写内容 M 的语句，即为攻击发生点，也就是需要定位的错误根源。回溯算法如图 3 所示。

```
while(!ls_empty(log))
{
  entry= get_prev_log_entry();
  if ( entry.write_address<=a2
       && entry.write_address+entry.len > a2 )
  {
       if(entry.data[a2-entry.write_address]== M )
       {
       get(entry.file_name, entry,line_number);
       break;
       }
  }
}
```

图 3 定位源程序漏洞的回溯算法

最后，我们使用原型系统对几个真实的带有漏洞的 GNU 工具软件进行漏洞攻击的错误定位。我们的测试平台是 kubuntu 7.04 Kernel 2.6.20，Intel CPU 1.9 GHZ，786M RAM。

我们选择的 3 个带有攻击样例的漏洞软件如表 2 所示。

表 2 带有漏洞的软件和漏洞描述

CVE#	Program	Vulnerability Description
CVE-2002-1549	lhttpd 0.1	Buffer overflow in Log function in util.c
CVE-2003-0466	Wu-ftpd 2.6.2	Buffer overfow in the fb_realpath function in realpath.c
CVE-2002-1904	ghttpd 1.4	Buffer overflow in Log function in util.c

经过测试后，我们的系统成功定位到了这些攻击的程序漏洞的源程序行。试验表明，研究具体的程序语义与可能的软件漏洞攻击点之间的关系，有针对性地实施漏洞/错误定位方法，并通过植入广义的软件安全代码获得运行时刻的攻击标签，进而对漏洞定位甚至消除，在很大程度上得到比已有工作更好的结果。

但是我们的方法无法定位间接攻击的漏洞或者漏洞根源并不是由于这些库函数的调用。例如间接攻击先通过缓冲区溢出，覆盖某个指针，然后通过程序内部本来的对该指针的赋值，而改变攻击目标的值，从而达到漏洞攻击的目的。由于我们只对不安全的库函数做内存更新记录，因此无法定位这些程序漏洞。更通用的内存更新记录方法需要进一步的研究，也是我们下阶段的主要工作。

4 相关工作

长期以来，针对恶意攻击的软件安全技术的研究主要集中在攻击检测方面，近期很多研究工作已经开始关注软件完整的攻击保护。例如利用攻击特征生成和过滤技术，以及程序的回退和恢复技术来保护受攻击的程序。这些工作的成效取决于对软件漏洞的认知程度和定位精度，因此部分工作提出了针对漏洞攻击的定位方法。Covers[15]提出一种定位漏洞攻击恶意输入的方法，目的是在漏洞攻击程序发生错误后定位攻击输入，从而生成对应的攻击特征(signature)；Dira[18]提出一种程序漏洞定位方法，同样以定位恶意输入为目的，定位导致程序控制流对象腐败的输入数据包，以便通过输入内容过滤，阻止相同的攻击再次发生。内存更新记录在 Dira 中用于程序回退执行，而缺乏对漏洞本身的定位分析。TaintCheck[19]利用程序的数据流信息来进行污染分析，定位恶意输入导致漏洞攻击错误的网络包。这些定位并没有真正定位到程序中漏洞所在，而是关注如何定位与攻击相关的网络包输入，在网络层提供基于攻击特征的输入过滤，这样只能提供对包含漏洞软件的临时性保护，无法从根本上消除软件漏洞。XunJun 等在文献[16]提出了针对程序漏洞的自动诊断方法，其目标是定位导致内存腐败的指令。由于缺乏审计信息，需要多次运行来定位指令，而且指令级别的定位对应到源程序语句很多情况下是库函数内部的语句，并不是本地代码的调用点。我们的工作在运行时动态记录广义的审计信息，只需要运行一次就能精确定位到漏洞源程序行。AutoPag[8]利用边界检查(Boundary checking)辅助定位于程序漏洞定位，并生成虚拟补丁，提供完整的攻击防护。但它只能够处理缓冲区溢出攻击，对其他的攻击方式，比如格式字符串攻击无能为力。

结束语 本文提出了一种基于内存更新记录的漏洞攻击错误定位方法，能直接定位到程序漏洞源程序行。通过测试，我们的系统精确定位到了利用库函数造成缓冲区溢出和格式化字符串溢出攻击的程序漏洞。

虽然我们定位了错误(程序漏洞)源程序语句，但还没有提出一套完整的方法来处理该漏洞，仍需要人工干预来生成补丁。因此，基于程序分析，具有程序意识地研究程序中各种安全相关的对象静态和动态的特征，建立完整的攻击/错误检查与发现、容忍与消除以及能够主动反馈的安全保障体系成为我们将来的研究目标。

参 考 文 献

[1] McGraw G. Software Security. IEEE Security & Privacy, 2004, 2(2): 80-83

[2] One A. Smashing The Stack For Fun And Profit. Phrack, 1996, 7(49)

[3] Argamal L. Ftpd: the advisory version. bugtraq mailing list, 23 June 2000. http://www.securityfocus.com/archive/1/66544

[4] Kaempf M. Vudo malloc tricks. http://www.phrack.org/phrack/57/p57-0x08

[5] Cowan C. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks // 7th USENIX Security Conference. San Antonio, Texas, USA, 1998

[6] Chiueh T, et al. RAD: A Compile-time Solution to Buffer Overflow Attacks // Proceedings of The 21st IEEE International Conference on Distributed Computing System. April 2001

[7] The PaX team. http://pax.grsecurity.net

[8] Lin Zhiqiang, et al. AutoPaG: Towards Automated Software Patch Generation with source. ASIACCS, 2007

[9] Korel B, Laski J. Dynamic program slicing. 1988

[10] Zellerand A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002, 28(2): 183-200

[11] Cleve H, Zeller A. Locating Causes of Program Failures // ICSE 2005. St. Louis, Missouri, May 2005

[12] Light HTTPD GET Request Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/6162

[13] Ghttpd Log Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960

[14] Multiple Vendor C Library realpath() Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/8315

[15] Liang Z, Sekar R. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers // CCS'05. Nov. 2005

[16] Xu J, et al. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities // CCS'05. Nov. 2005

[17] http://hal.cs.berkeley.edu/cil/

[18] Smirnov A, et al. DIRA: Automatic Detection, Identification, and Repair of Control-hijacking Attacks // Proc. of NDSS'05. San Diego, CA, 2005

[19] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software // NDSS'05. Feb. 2005

[20] 夏耐，郭明松，茅兵，等. 基于简化控制流监控的程序入侵检测. 电子学报，2007，35(2)：358-361