ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.2, February 2010, pp.179−193
doi: 10.3724/SP.J.1001.2010.03785

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

面向高可信软件的整数溢出错误的自动化测试*

卢锡城, 李 根+, 卢 凯, 张 英

(国防科学技术大学 计算机学院,湖南 长沙 410073)

High-Trusted-Software-Oriented Automatic Testing for Integer Overflow Bugs

LU Xi-Cheng, LI Gen+, LU Kai, ZHANG Ying

(School of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: superligen@gmail.com

Lu XC, Li G, Lu K, Zhang Y. High-Trusted-Software-Oriented automatic testing for integer overflow bugs. *Journal of Software*, 2010,21(2):179−193. http://www.jos.org.cn/1000-9825/3785.htm

Abstract: This paper presents an automatic testing method, DAIDT (dynamic automatic integer-overflow detection and testing), for finding integer overflow fatal bugs in binary code. DAIDT can thoroughly test the binary code and automatically find unknown integer overflow bugs without necessarily knowing their symbol tables. It is formally proved in this paper that DAIDT can theoretically detect all the high-risk integer overflow bugs with no false positives and no false negatives. In additional, any bugs find by DAIDT can be replayed. To demonstrate the effectiveness of this theory, IntHunter has been implemented. It has found 4 new high risk integer overflow bugs in the latest releases of three high-trusted applications (two Microsoft WINS services in Windows 2000 and 2003 Server, Baidu Hi Instant Messager) by testing each for 24 hours. Three of these bugs allow arbitrary code execution and have received confirmed vulnerabilities numbers, CVE-2009-1923, CVE-2009-1924 from Microsoft Security Response Center and CVE-2008-6444 from Baidu.

Key words: integer overflow; integer overflow vulnerability; dynamic automatic test case generation; taint analysis; symbolic execution

摘 要: 面向高可信软件提出了一种二进制级高危整数溢出错误的全自动测试方法(dynamic automatic integer-overflow detection and testing,简称 DAIDT).该方法无需任何源码甚至是符号表支持,即可对二进制应用程序进行全面测试,并自动发现高危整数溢出错误.在理论上形式化证明了该技术对高危整数溢出错误测试与发掘的无漏报性、零误报性与错误可重现特性.为了验证该方法的有效性,实现了 IntHunter 原型系统.IntHunter 对 3 个最新版本的高可信应用程序(微软公司 Windows 2003 和 2000 Server 的 WINS 服务、百度公司的即时通讯软件 BaiDu Hi)分别进行了 24 小时测试,共发现了 4 个高危整数溢出错误.其中 3 个错误可导致任意代码执行,其中两个由微软安全响应中心分配漏洞编号 CVE-2009-1923,CVE-2009-1924,另一个由百度公司分配漏洞编号 CVE-2008-6444.

关键词: 整数溢出;高危整数溢出错误;动态自动测试用例生成;污点分析;符号化执行

中图法分类号: TP311 **文献标识码**: A

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2007AA010301 (国家高技术研究发展计划(863)); the National Basic Research Program of China under Grant No.2005CB321801 (国家重点基础研究发展计划(973))

Received 2009-06-15; Revised 2009-09-11; Accepted 2009-12-07

随着计算机应用的不断发展,软件已渗透到国民经济和国防建设的各个领域,高可信软件以其较高的可靠性(reliability)、安全性(safety)等特征[1]成为国家信息基础设施的重要组成部分,在信息社会中发挥着至关重要的作用.因此,高可信软件中存在的任何潜在错误与漏洞都将导致生命财产的重大损失或者对周围环境造成严重破坏.

在众多的已知软件代码错误种类中,高危整数溢出错误是一类隐蔽极深的错误,多发生于出现概率小却可能发生的程序极端情况或极端数据的处理过程中,并由于其隐蔽性,即使在经过严格测试的高可信软件中仍有可能存在高危整数溢出错误.例如,1996 年 6 月 4 日,在欧洲 Ariane 5 火箭初次发射时,由于一个 64 位浮点数转换成 16 位有符号整数时产生了整数溢出,使得火箭控制系统错误地发出控制指令,最终导致火箭腾空爆炸的灾难性后果.

图 1 给出了近年来国际权威漏洞披露组织CVE(common vulnerabilities and exposures)发布的高危整数溢出错误的历年数量,由于当前缺乏对此类错误的有效测试发掘技术,该类错误的数量呈现逐年递增的趋势.图 2 给出了CVE发布的全球 2006 年软件代码安全漏洞数量的统计分布情况[2].从图中可以看出,整数溢出错误已成为仅次于缓冲区溢出错误的威胁软件安全的第二大类错误.因此,如何全面、有效、精确地检测出大规模软件中潜在的高危整数溢出错误,已成为近年来研究者们关注的焦点.

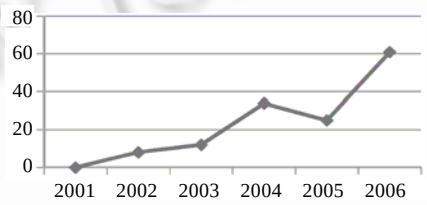


Fig.1 Number of high risk integer-overflow bugs disclosed by CVE from 2001 to 2006

图 1 CVE 发布的 2001 年~2006 年的高危整数溢出错误的数量

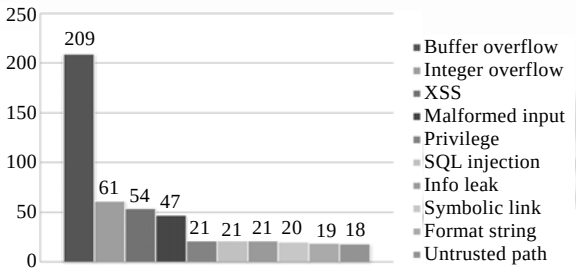


Fig.2 Number of the top-ten software bugs disclosed by CVE in 2006

图 2 CVE 发布的 2006 年前十大软件错误的数量

为了全面、有效地发掘软件中潜在的高危整数溢出错误,本文提出了一种在二进制级对高可信软件进行全面高危整数溢出错误发掘的自动化动态测试方法——DAIDT(dynamic automatic integer-overflow detection and testing).DAIDT 方法无须程序员编写任何测试代码或设计任何测试用例,在无任何源码甚至是符号表的情况下,可以全自动地对目标软件进行无重复的全面路径覆盖测试,以准确定位并有效发掘可能导致严重安全事故的高危整数溢出错误.DAIDT 的整数溢出确认和定位技术可以保证零误报.此外,由于 DAIDT 为每个高危整数溢出错误所在的指令执行路径都生成对应的测试用例,从而可以辅助开发或测试人员重现错误.

为了验证 DAIDT 方法的有效性,我们实现了原型系统 IntHunter.该系统分别对微软公司的最新版本 Windows 2003 Server 的 WINS 服务(按照高可信度计算标准开发)、最新版本 Windows 2000 Server 的 WINS 服务以及百度公司即时通信软件 Baidu Hi 进行测试,一共发现新的高危整数溢出漏洞 4 处,并且均得到官方承认.实验结果验证了 DAIDT 方法的有效性.

1 基本概念

首先区分整数溢出与高危整数溢出错误这两个概念.

定义 1(整数溢出). 在计算机程序中,一个有/无符号整数通常由一组固定位宽的位向量表示(通常是 64 位、32 位、16 位或 8 位).在程序指令操作过程中,如果运算结果的整数数值超出了存储结果的位向量所能表达的整数范围,则该指令操作发生了整数溢出.

事实上,整数溢出本身并不会直接对可信软件的安全性与可靠性构成威胁,甚至不一定是一种错误.比如,程序员可能为了提取低位数据,故意使用左移操作使得整数高位数据溢出.我们通过对超过 200 个已知严重的整数溢出漏洞案例进行调查研究发现,对软件安全产生危害的整数溢出通常通过旁路路径约束检查,从而抵达一般条件下无法抵达的循环、指针操作等“可信”代码段,通过控制它们来产生威胁软件安全的高危错误.因此,我们归纳总结出这些高危整数溢出错误的共性,并给出其定义.

定义 2(高危整数溢出错误). 由极端输入数据引起的整数溢出旁路了程序的正常状态检查条件,进而使得接下来的“可信”程序代码段在错误输入数据的控制下,最终发生程序员没有处理(也没有预料到)的错误.

根据定义 2 可知,高危整数溢出错误具有如下性质:

(1) 高危整数溢出错误一定出现在一条由输入数据控制的可行指令执行路径上;

(2) 高危整数溢出错误通常由具有极端数值的输入数据引发;

(3) 高危整数溢出错误必须引发程序员没有考虑到且没有处理的程序异常.

可见,如果高可信软件中的该类错误被恶意利用,则可能引发高可信软件崩溃并丧失服务能力,甚至可能被远程执行任意代码,进而被全面控制,丧失可信性.

2 DAIDT:精确高危整数溢出错误的自动化测试技术

如上所述,高危整数溢出是一种路径相关且发生于特定执行路径上的高危软件错误.理论上,如果能够完全遍历程序中所有与输入相关的执行路径,就能够发现所有的高危整数溢出错误.实现这一目标需要解决两个问题: (1) 如何准确指导程序覆盖所有与输入相关的可行指令路径,以发现所有的高危整数溢出错误;(2) 如何在有限的测试时间内尽早地暴露高危整数溢出错误.其中,第 2 个问题的提出是出于以下考虑:在自动化测试时,假设程序输入数据为n个字节(每字节 8 位),那么程序中处理输入数据的可行路径数量最多可达2^{8n}条,然而实际测试中,测试时间有限,因此需要一种有效的启发算法,能够将最有可能发生高危整数溢出错误的路径提前暴露.

为了解决第 1 个问题,DAIDT 将输入数据字节视为符号化污点,通过动态追踪这些符号化污点在程序中的传播情况,收集程序执行路径对输入数据(污点)的约束,求解并生成新的测试用例,并用不断产生的新的测试用例指导程序对其自身进行全面路径覆盖,我们将在第 2.1 节进行更详细的讨论.同时,为了解决第 2 个问题,DAIDT 在全面路径搜索的基础上,通过面向高危整数溢出错误发掘的测试用例调度技术,尽量在有限时间内,有针对性地将最有可能发生高危整数溢出错误的测试用例提前调度,以尽早暴露并自动捕捉高危整数溢出错误所在的可行路径.我们将在第 2.3 节进行更详细的讨论.

2.1 DAIDT的测试用例生成技术

二进制程序对输入数据的处理过程可形式化地定义为标记迁移系统(labeled transition system,简称LTS): $BP=\langle S,s_1,T,\Delta\rangle$,其中,$S$是程序指令集合,$s_1\in S$是程序在接收输入数据后执行的第 1 条二进制指令,T是迁移集合,$\Delta=S\times T\times S$表示迁移关系.在程序动态执行过程中,存在迁移关系$\langle s_1,t,s_2\rangle\in\Delta$,其中,迁移$t$定义如下:

如果s_1是赋值语句(assignment),$t=s_1$,或者如果s_1是分支语句“if (c) s_A; else s_B;”,那么,

$$t=\begin{cases} assume(c), & \text{if } s_2=s_A \\ assume(\neg c), & \text{if } s_2=s_B \end{cases}.$$

对于一个程序所对应的标记迁移系统$BP=\langle S,s_0,T,\Delta\rangle$来说,迁移集合$T$中只包含两种语句:赋值语句和assume

语句.其中,assume语句对应的迁移描述了系统迁移到下一个状态必须满足的条件.一条从 1 时刻起开始执行到 i 时刻的执行路径可以表示为一系列指令迁移构成的序列,记为 $P_{t_i}=t_0,...,t_i$.

根据Hoare逻辑,程序语义可以基于最强后置条件给出[3].为了能够更加准确地描述和讨论动态收集的具体程序执行路径的最强后置条件,我们借鉴文献[4]使用二元组$\langle\Omega,\Phi\rangle$表示和计算最强后置条件.其中,$\Omega$是一个部分函数,将程序变量映射到由 Skolem 常量组成的表达式,即Ω:$Vars\mapsto Exp$($Vars$ 和 Exp 分别表示程序变量和表达式的全集).我们可以按照通常的方法将Ω提升到表达式集合,Ω:$Exp\mapsto Exp$;Φ是包含了所有assume语句引入的布尔表达式的集合.基于$\langle\Omega,\Phi\rangle$的赋值语句和assume语句的最强后置条件的计算方法定义如下[4]:

$$SP(x:=e)=\lambda\langle\Omega,\Phi\rangle.\langle\Omega[x\to\Omega(e)],\Phi\rangle,$$

$$SP(assume(c))=\lambda\langle\Omega,\Phi\rangle.\langle\Omega,\Phi\cup\Omega(e)\rangle,$$

其中,函数$\Omega[x\to e]$定义为

$$\Omega[x\to e](y)=\begin{cases}\Omega(y), & \text{if } y\neq x\\ e, & \text{if } y=x\end{cases}.$$

经典的最强后置条件给出了程序的精确语义,即描述了执行某个程序语句后变量取值所满足的条件.而在动态污点传播与动态测试用例生成过程中,只需关注程序对输入数据的直接与间接处理过程,并对该部分程序行为进行抽象.在DAIDT中,我们将m字节输入数据逐字节地定义为直接符号化污点变量(尚未参与程序中任何操作的初始变量),并将由输入数据直接或间接计算的变量定义为间接符号化污点变量,所有符号化污点变量保存在污点变量集合Θ中.在程序初始状态时,Θ只包含直接符号化污点变量,即$\Theta=\{\theta_1,\ldots,\theta_m\}$,$0\le\theta_i\le 0xff$.DAIDT通过与污点变量相关的部分最强后置条件对程序语义进行近似,即只关注污点变量集合Θ,通过动态污点追踪,产生与污点变量相关的部分最强后置条件,记为SP_Θ.

在讨论二进制级程序执行路径的部分最强后置条件SP_Θ的动态构造与收集过程之前,我们首先对二进制程序模型进行一些基本假设:

1. 程序内存空间可以抽象成一个一维数组 $\mathcal{M}$;
2. 任何程序变量都存在位宽限制,且只能存储于 $\mathcal{M}$ 或确定位宽的寄存器 $\mathcal{R}$ 中;
3. 任何CPU指令语义可以等价的转换成为仅由赋值语句构成的代数表达式[5],其中,代数表达式仅由$\mathcal{M}$或$\mathcal{R}$参与.

例如,指令 mov eax,[ebx+4]可等价转换为 $eax=\mathcal{M}[ebx+4]$,条件分支指令 jz 0x100 可等价转换为 $eip=(ZF!=0)\ ?\ 0x100:nextIP$,其中,$ZF$ 为标志位寄存器,$nextIP$ 为当前指令的下一条指令的地址,(ZF!=0)是状态迁移系统中 assume 语句所使用的条件 c.

在对SP_Θ的动态构造与收集过程中,需要对CPU中流过的指令的迁移t_i进行动态的符号化污点传播计算,过程如下:

首先,对迁移t_i的右表达式进行符号化替换:

(a) 用直接或间接符号化污点变量替换相应的操作数.

(b) 用具体的运行时信息(如寄存器值等)替换对应的、与符号化污点无关的操作数.

(c) 对于表达式中的内存访问 $\mathcal{M}[x]$,在线地计算当前访存的具体地址,如果该地址处内容存在符号化污点,则用符号化污点变量替换 $\mathcal{M}[x]$;否则,用该地址中的具体数据替换 $\mathcal{M}[x]$.

然后,如果符号化替换后的右表达中包含符号化污点变量,则生成新符号化污点变量θ_{m+i}(i为指令迁移t_i的下标).

最后,将符号化污点变量θ_{m+i}赋值给指令迁移t_i的目的操作数,并更新污点变量集合$\Theta=\Theta\cup\theta_{m+i}$.

上述过程完成了符号化污点传播,并且该传播过程具备单赋值性质.例如,指令mov eax,ebx+4 对应的指令迁移t_1的表达式为$eax=ebx+4$,若ebx存有符号化污点变量θ_1,则替换ebx+4 为θ_1+4,并产生新的符号化污点$\theta_{m+1}=\theta_1+4$,最后将θ_{m+1}赋值给eax目的寄存器eax,即$eax=\theta_{m+1}=\theta_1+4$.

根据符号化污点传播过程,我们讨论只与污点相关的部分最强后置条件的构造.记函数$Vars(e)$返回表达式e

中所有污点变量构成的集合,例如,$Vars(\theta_1+4\times\theta_2)=\{\theta_1,\theta_2\}$.定义赋值语句$x:=e$在上述符号化污点传播后形成的部分最强后置条件$SP_\Theta(x:=e)$为:如果$Vars(e)\subseteq\Theta$,则$SP_\Theta(x:=e)=SP(x:=e)$;否则,$SP_\Theta(x:=e)$是一个恒等函数.定义assume语句$assume(c)$在上述符号化污点传播后形成的部分最强后置条件$SP_\Theta(assume(c))$为:如果$Vars(e)\subseteq\Theta$,则$SP_\Theta(assume(c))=SP(assume(c))$;否则,$SP_\Theta(assume(c))$是一个恒等函数.对于程序执行路径$P_{t_n}=t_1,t_2,\ldots,t_n$,$SP_\Theta(P_{t_n})=SP_\Theta(t_1)\circ SP_\Theta(t_2)\circ\ldots\circ SP_\Theta(t_n)$,其中,"$\circ$"定义为从右向左组合.

根据与污点相关的部分最强后置条件,我们可以构造符号化路径约束.令当前路径为$P_{t_n}=t_1,\ldots,t_n$,通过定义如下函数$PathCond$,可将SP_Θ的二元表示$\langle\Omega,\Phi\rangle$转换为一阶逻辑表达式:

$$PathCond(p_{t_n})=PathCond(\langle\Omega,\Phi\rangle)=\exists\theta_1\ldots\theta_{m+n}((\bigwedge_{\omega\in\Omega}e2b(\omega))\wedge(\bigwedge_{\phi\in\Phi}\phi)),$$

其中$\Theta=\{\theta_1\ldots\theta_{m+n}\}$,$\{\theta_1\ldots\theta_m\}\subseteq\Theta$是初始输入数据集合,即直接符号化污点变量集合,$\Theta-\{\theta_1\ldots\theta_m\}=\{\theta_{m+1}\ldots\theta_{m+n}\}$是污点传播过程产生的间接符号化污点变量集合.函数$e2b(\omega)$将部分函数$\Omega$的每个变量或表达式映射转换为一个布尔表达式,例如,$e2b(\theta_i:=\theta_1+4)=(\theta_i==(\theta_1+4))$.

根据文献[4],若路径P_{t_n}是可行的,那么$PathCond(P_{t_n})$=true,反之也成立.此外,如果路径P是可行的,那么一定可以构造合适的数据集合$\Theta'=\{\theta'_1\ldots\theta'_{m+n}\}$使得$PathCond(P_{t_n})$=true.提取其中的$\{\theta'_1\ldots\theta'_m\}\subseteq\Theta$来构造输入数据$\Psi$,那么输入数据$\Psi$将可以指导程序从指令迁移$t_1$执行到当前的指令迁移$t_n$.

引理 1. 从任意一组输入数据$\Theta=\{\theta_1,\ldots,\theta_m\}$出发,若其执行路径上的存在一个受污点控制的分支指令的迁移$t_i=assume(c)$(即$Vars(e)\subseteq\Theta$),若存在一组数据集合$\Theta'=\{\theta'_1,\ldots,\theta'_{m+i}\}$满足$(\neg c)\wedge PathCond(P_{t_{i-1}})$,那么新输入数据集$\{\theta'_1\ldots\theta'_m\}\subseteq\Theta$可以指导指令迁移$t_i$向$\neg c$方向迁移.

证明:若存在新输入数据集合$\Theta'=\{\theta'_1,\ldots,\theta'_{m+i}\}$满足$(\neg c)\wedge PathCond(P_{t_{i-1}})$,由$PathCond(P_{t_{i-1}})=$true可知,新输入数据对应的执行路径$P'$中,其前$i-1$条指令一定与$P$中前$i-1$条指令相同.根据状态迁移指令的定义可知,新的$t'_i=assume(\neg c)$,状态迁移指令的迁移方向与$t_i=assume(c)$相反. □

根据引理 1,DAIDT通过测试用例生成指导路径覆盖的方法可以描述为:当程序在初始输入数据指导下执行,每遇到一个受污点变量控制的分支指令时,记当前分支指令迁移$t_n=assume(c)$.那么,要让分支向另一方向跳转,只需计算路径约束$(\neg c)\wedge PathCond(P_{t_{n-1}})$,若存在可满足解,就可构造一个新的测试用例$\Psi$,该测试用例可以指导程序从$t_1$运行至$t_n$,并指导下一条指令向$\neg c$的方向跳转.当一个测试输入数据被处理完毕后,DAIDT将程序恢复到初始状态,根据测试用例调度规则,继续送入第2个测试用例数据.不断迭代上述过程,DAIDT可以无重复地遍历程序中所有与输入相关的可行路径.

下面讨论DAIDT基于测试用例生成指导路径覆盖来发掘高危整数溢出错误的原理与方法.讨论将基于如下基本假设:(1) 待检测的目标高可信程序中不存在软件逻辑或功能性错误,这可以通过前期的模型验证及后期的系统功能测试等技术得到保证.(2) 实际程序设计时,程序员容易忽略整数位宽这一隐含限制,而基于一个理想的无位宽限制(即没有整数溢出)的机器模型来设计程序.(3) 在无位宽限制条件下,程序员定义的严格、合理的分支路径约束,可以对路径P上每一条指令t_i执行结果取值范围产生精确约束,以保证程序的正确执行.

我们将在满足路径约束情况下、指令迁移表达式t_i在无位宽限制的理想机器模型中所有可能结果的集合定义为$\mathcal{R}^P_{ideal}(t_i)$;而将$t_i$实际在执行中可能产生的所有真实执行结果的实际取值集合定义为$\mathcal{R}^P_{real}(t_i)$.在实际执行中,由于位宽限制的存在,$t_i$的一些非程序员的预期值参与算术操作后,因发生整数溢出,而使其执行结果满足了程序员的路径约束,因此$\mathcal{R}^P_{real}(t_i)\supseteq\mathcal{R}^P_{ideal}(t_i)$,如图3所示.也就是说,$\mathcal{R}^P_{real}(t_i)$既包含了不发生整数溢出的结果集合,也包含了由于程序员疏忽,通过发生整数溢出而满足程序员约束的结果集合,这些结果都不在程序员预期范围内.例如,对于无符号整数变量x,令指令迁移t_i为$x=e(\ldots)$,当分支条件为$(x*5<=5)$时,在无位宽限制的情况下,t_i指令的值集$\mathcal{R}^P_{ideal}(t_i)$取值范围(即$x$的取值范围)限制为$\{0,1\}$;然而在32位位宽限制时,存在如$0xccccccccd*5=1$等整数溢出情况,则

$$\mathcal{R}^P_{real}(t_i)=\{0,1,0xccccccccd,\ldots\}\supset\mathcal{R}^P_{ideal}(t_i).$$

定义集合$\mathcal{R}^P_{except}(t_i)=\mathcal{R}^P_{real}(t_i)-\mathcal{R}^P_{ideal}(t_i)$,则$\mathcal{R}^P_{except}(t_i)\neq\varnothing$时,存在整数溢出.需要注意的是,$\mathcal{R}^P_{ideal}(t_i)$和$\mathcal{R}^P_{real}(t_i)$

均为几乎不可计算的两个集合,引入它们仅仅是为了方便讨论.

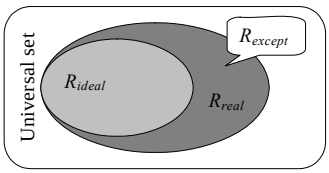


Fig.3 Relationship of R_{ideal}, R_{real} and R_{except}

图 3 R_{ideal},R_{real}和R_{except}的相互关系

本质上,若由指令迁移t_n的值参与计算的某条指令迁移表达式t_{n+j}存在整数溢出,那么当程序执行到指令迁移t_{n+j}后的某一分支指令迁移t_{n+j+k}:$assume(c)$时,DAIDT将为其生成新的测试用例并构建路径约束$\neg c \wedge PathCond(P_{t_{n+j+k-1}})$,若约束足以将$\mathcal{R}^P_{real}(t_n)$集合中的$\mathcal{R}^P_{ideal}(t_n)$部分完全排除,那么它将迫使求解器在为$t_{n+j+k}$分支寻求可满足解时,只能在$\mathcal{R}^P_{except}(t_n)$部分寻找可满足解.如果可满足解存在,DAIDT就找到并构造出了一个能够使得程序执行到t_{n+j}时产生整数溢出的测试用例.

定义 3. $PathCond_{bv}(P_{t_i})$表示在有位宽限制的真实机器中,通过污点传播收集到的当前路径约束,该路径约束中的变量由具有确定位宽的位向量表示.

定义 4. $PathCond_{ideal}(P_{t_i})$表示在无位宽限制的理想机器中,通过污点传播收集到的当前路径P_{t_i}的路径约束,该路径约束中的变量由无位宽限制的变量表示.

在实际使用中,$PathCond_{ideal}(P_{t_i})$可通过将$PathCond_{bv}(P_{t_i})$中每一个变量的位宽扩大 2 倍来近似地获得.

定理 1. $PathCond_{ideal}(P_{t_i}) \Rightarrow PathCond_{bv}(P_{t_i})$,反之则未必.

证明:根据基本假设 1 与基本假设 3,由于程序不存在逻辑或功能性错误,那么在无位宽限制条件下,程序员定义的严格、合理的分支路径约束$PathCond_{ideal}(P_{t_i})$=true 时,表示路径$P_{t_i}$在实际执行过程中是可行的.根据前文所述,若路径可行,则实际执行时收集到的位精确路径约束$PathCond_{bv}(P_{t_i})$一定为真.因此,$PathCond_{ideal}(P_{t_i}) \Rightarrow PathCond_{bv}(P_{t_i})$成立.

反之,若路径P_{t_i}上存在高危整数溢出错误,则根据高危整数溢出定义,由于整数溢出可以旁路$PathCond_{bv}(P_{t_i})$中的相关约束,仍然可以使得$PathCond_{bv}(P_{t_i})$=true;然而根据基本假设 3,在无位宽限制情况下,程序员设计的路径约束$PathCond_{ideal}(P_{t_i})$是完备的,这意味着整数溢出无法通过$PathCond_{ideal}(P_{t_i})$条件约束,即$PathCond_{ideal}(P_{t_i})$=false. □

定理 2. 若路径P中存在指令迁移t_i且$\mathcal{R}^P_{except}(t_i) \neq \varnothing$,则程序执行路径$P$上存在整数溢出.

证明:用反证法容易证明,在此不再赘述. □

定理 3. 若程序中存在高危整数溢出错误,则一定存在一条可行路径P,满足$PathCond_{bv}(P)$=true 且$PathCond_{ideal}(P)$=false.

证明:根据定理 1 的证明可知. □

定理 4. 在高可信程序中,从一个正常输入数据出发,按照 DAIDT 路径覆盖的方法描述,可以自动化地完全发掘与输入相关的高危整数溢出错误.

证明:若程序中存在高危整数溢出错误,则根据定理 3 可知,一定存在$PathCond_{bv}(P)$=true 且$PathCond_{ideal}(P)$=false,即一定存在一条包含该错误的与输入相关的可行路径P.再根据DAIDT路径覆盖的方法描述可知,测试系统可以通过生成测试用例无重复的覆盖t_1开始的所有与输入相关的可行路径,而这其中一定包含了该可行路径P.当程序在测试用例指导执行下触发了未知异常,依据定理 2 能够自动化地判定高危整数溢出错误. □

2.2 实例分析

我们以图 4 中包含高危整数溢出错误的代码为例,实例化地介绍第 2.1 节所述的 DAIDT 的形式化方法.

图 4 中,代码首先读入一个 32 位输入数据到 *eax* 中(第 1 行),该输入数据表示待拷贝数据元素的个数(其中每个数据元素为 5 字节);然后,在进入循环拷贝之前,对该输入数据表示的待拷贝数据元素个数进行长度检查(第 2 行~6 行),即 *eax**5 bytes/element 必须大于 0 且小于等于 *ecx*(实际运行时,*ecx*=5);最后进入到循环,按照输入数据指定的次数进行循环拷贝(第 7~9 行).

从代码可以看出,程序员希望通过条件约束 $0<eax*5\le ecx$ 将输入数据控制在合理范围.在实际执行中,与输入数据无关的寄存器 *ecx* 的值为 5,因此只有当输入数据为 1 时,*eax*=1,条件 $0<(eax*5)\le 5$ 才能得到满足,从而顺利通过条件检查进入循环拷贝.由于程序员没有考虑乘法运算带来的潜在整数溢出,当示例代码运行一遍循环体后,至循环出口(第 9 行)时,程序员认定 *eax* 的结果值集合只能为 $\mathcal{R}^{P}_{ideal}(t_8)=\{0\}$(当执行到第 9 行时,*eax* 的值来自于指令迁移t_8).此时,第 9 行的分支指令满足跳出循环的条件.因此,程序将选择跳出循环.

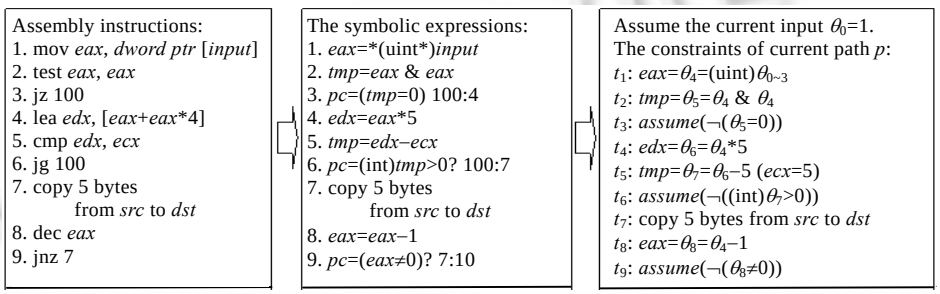


Fig.4 A high risk integer overflow case

图 4 高危整数溢出实例

真实执行中,示例代码存在整数溢出.以$\theta_{0\sim3}$作为输入数据变量,则当$\theta_{0\sim3}$=0*xcccccccd*时,

$$\theta_{0\sim3}*5=0xcccccccd*5=0x1.$$

尽管程序员希望在第 3 行和第 6 行处通过严格的条件约束($0<eax*5\le ecx$)来控制循环次数,防止拷贝越界.但由于存在整数溢出,第 3 行和第 6 行约束检查被旁路,最终进入到循环,引起循环执行 0*xcccccccd* 次.这远超过程序员设计的最大理想值(1),发生了因循环次数过量导致的拷贝越界,从而产生高危整数溢出错误.

DAIDT方法对该高危整数溢出错误的发掘过程如下:设输入数据$\theta_{0\sim3}$=0*x*1,DAIDT以该合法输入数据为起始搜索程序路径,动态地进行符号化污点传播.图 4 中右图给出了示例代码执行到第 9 行时,DAIDT构建出符号化污点路径约束表示.在进行路径搜索时,DAIDT将每个状态迁移指令*assume*(*c*)中的条件*c*取反,求解并构造可满足进入¬*c* 分支的输入数据.当示例代码执行到循环出口(第 9 行)时, $PathCond_{bv}(P_{t_9})$ 化简为

$$(\theta_{0\sim3}\neq0)\wedge((\text{uint32})\theta_{0\sim3}*5\le5)\wedge((\theta_{0\sim3}-1)==0).$$

按照程序员的预先设计,这些条件约束使得控制循环次数的寄存器*eax*($eax=\theta_8=\theta_4-1=\theta_{0\sim3}-1$)的值在此之后只能为 0,即 $\mathcal{R}^{P}_{ideal}(t_8)=\{0\}$,示例代码已经不可能继续从第 9 行再次进入循环.当 DAIDT 执行到第 9 行时,将指令迁移t_9中的条件*c*(即$(\theta_{0\sim3}-1)==0$)取反得$(\theta_{0\sim3}-1)\neq0$,并联合之前第 9 行之前的路径约束,得到一组新的约束$((\theta_{0\sim3}-1)\neq0)\wedge(\theta_{0\sim3}\neq0)\wedge((\text{uint32})\theta_{0\sim3}*5\le5)$,满足该约束的解将使示例代码从第 9 行再次进入循环,这相当于逼迫求解器给出一组能够让循环继续下去的可满足约束的输入数据.求解器在该约束下将可能生成$\theta_{0\sim3}$=0*xcccccccd*,$\theta_{0\sim3}$=0*x*33333334,$\theta_{0\sim3}$=0*x*66666667,$\theta_{0\sim3}$=0*x*9999999*a*中的任何一个.这些输入数据在第 4 行处产生整数溢出,并且使得第 8 行的异常结果集合 $\mathcal{R}^{P}_{except}(t_8)=\{0xcccccccc,0x33333333,0x66666666,0x99999999\}\neq\varnothing$.将数据 0*xcccccccd* 再次作为输入数据送入目标代码后,程序必将产生异常.DAIDT 捕捉到程序未知异常后,根据定理 2 确认为高危整数溢出错误.

值得注意的是,DAIDT 对高危整数溢出错误的有效发掘完全无需人工经验或是模板匹配,而是有目的地自

动搜索程序中所有可行路径,并在此过程中让不断收紧的程序路径约束 $PathCond_{bv}(P_{t_i})$ 将潜在整数溢出指令的理想结果集合 $\mathcal{R}^P_{ideal}(t_i)$ 完全排除,以迫使求解器只能在潜在的异常结果集 $\mathcal{R}^P_{except}(t_i)$ 中取值.若存在可满足解,则必定产生一条可行的整数溢出路径.

由于理论上DAIDT可以完全搜索出所有与输入相关的可行路径,因此如定理4所证,DAIDT能够完全发掘所有存在于程序中的高危整数溢出错误.此外,由于每条可行路径均对应于一组输入数据,DAIDT具备复现高危整数溢出错误的能力.

2.3 面向高危整数溢出错误发掘的启发式测试用例调度策略

如前所述,若程序输入数据最多为n个字节(每字节 8 位),那么程序中处理输入数据的可行路径数量最多可达 2^{8n}条.而实际测试过程中,测试时间是有限的,因此DAIDT采用启发式测试用例调度策略,将最有可能发生整数溢出的测试用例尽可能提前调度并进行动态追踪测试.

根据我们对超过 200 个整数溢出案例分析发现,按照以下启发式调度策略能够有效地在尽可能短的时间内发现高危整数溢出漏洞.

DAIDT为每个新产生的测试用例 Ψ_i 计算权重 ω_i,然后按照权重大的优先调度的原则进行测试(令 $\tau(i)$ 为DAIDT采用用例 Ψ_i 进行测试时可覆盖的新增基本块数量):

(a) 如果程序正进行指针操作且测试用例 Ψ_i 可以产生程序指针越界访问,则 ω_i=10000+$\tau(i)$;

(b) 如果程序执行到循环出口指令正准备跳出循环,且测试用例 Ψ_i 可以指导程序继续循环,则

$$\omega_i=500+\tau(i).$$

(c) 其他测试用例 Ψ_i, $\omega_i=\tau(i)$.

原则(a)表明,一个受输入控制的指针可能产生越界访问.在二进制代码中,指针可以看成是地址变量,当指令迁移表达式 t_i 产生的指针地址 $a \notin \mathcal{R}^P_{ideal}(t_i)$,即 $a \in \mathcal{R}^P_{except}(t_i)$ 时,即可认为指针a存在越界行为.若该程序满足上文提出的高可信程序编程基本假设,则根据定理 2 可知,该指针错误一定伴随整数溢出产生;然而是否属于高危整数溢出错误,则需要在真实运行后才能知晓.因此,这类型测试用例具有最高调度优先级.

对于原则(b),从如图 3 所示的伴随整数溢出的循环实例中可看到:当循环次数达到最大后,容易形成最强的路径约束 $PathCond_{bv}(P)$,使得整数溢出指令的理想结果集合 $\mathcal{R}^P_{ideal}(t_i)$ 被完全排除,从而迫使求解器只能在潜在的异常结果集 $\mathcal{R}^P_{except}(t_i)$ 中取值;若存在可满足解,则根据定理 2 必定存在一条可行的导致整数溢出的路径.因此,优先调度这类测试用例有助于提前发现高危整数溢出错误.

原则(c)则鼓励 DAIDT 优先调度那些预计对代码覆盖率贡献大的测试用例,因为优先调度这些测试用例有助于程序尽早地覆盖程序绝大部分代码,使得错误得以提前暴露.

3 系统原型与实验分析

3.1 系统原型

为了验证方法的有效性,我们实现了 DAIDT 的原型系统 IntHunter.图 5 给出了 IntHunter 的体系结构.该系统运行于具体执行待测试程序的全系统虚拟机之上,主要包括 4 个层次化部分:进程与线程追踪器、指令集体系结构相关的指令解码器与体系结构无关的指令表达式抽象映射器、符号化污点追踪与动态路径约束收集器和可满足求解器.下面自底向上简要介绍各个层次的功能.

进程与线程追踪器采用通用的进线程识别技术,定位当前 CPU 正在执行的进程及其线程,追踪器为上层动态分析提供精确控制任意一个进程/线程执行的能力,使得上层分析部分不必关心目标应用所运行的具体操作系统,且可以透明地对多种平台(如 Linux,Windows,FreeBSD 等)的应用进行统一的自动化追踪与测试.

指令集体系结构相关的指令解码与体系结构无关的指令表达式抽象映射层,负责动态地根据上层的需要,对目标进程中的部分指令进行解码,并映射成体系结构无关的指令表达式.抽象指令表达式可以适应不同体系

结构映射的需要,为 IntHunter 系统提供对面向多种指令集体系结构目标系统的扩展能力.

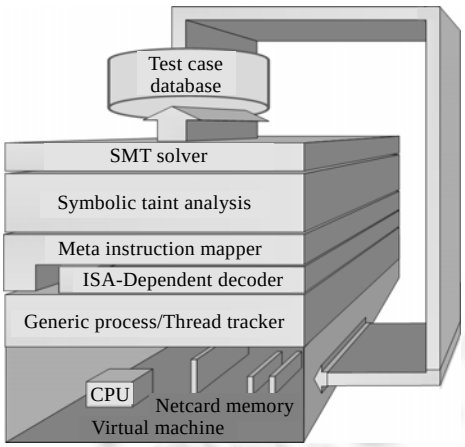


Fig.5 Architecture of IntHunter

图 5 IntHunter 的系统体系结构

符号化污点传播与动态路径约束收集器,将执行路径中与符号化污点输入数据相关的指令转换成对输入数据的约束条件.IntHunter系统采用可满足求解器[6]提供的位向量计算理论与数组理论,能够快速地、位精确地计算任意位向量约束.

可满足求解器对动态路径约束收集器收集的约束进行求解,当求解器得到可满足解后,便可以自动构造一个新的测试用例.

IntHunter 系统通过面向高危整数溢出错误的测试用例调度策略,为已生成的测试用例计算权重,并按从高到低的顺序依次调度,以动态地指导程序执行并搜索和遍历所有可行代码路径.若路径中存在满足定理 2 的指令且执行路径上发生了未知异常,那么 IntHunter 系统定位、报告并输出测试用例数据以辅助程序员重现该高危整数溢出错误.

3.2 实验分析

我们的测试平台硬件配置为 Intel E8400 CPU,内存为 4G,主机操作系统为 Vista SP1 x64.为避免 IntHunter 执行单个测试用例的时间过长,我们对测试时间作如下限定:

(1) 每个被调度的测试用例的分析与执行最长不超过 15 分钟,任何执行达到该时间边界的测试用例将自动停止,然后调度下一个测试用例执行.

(2) 每个待测程序的最大测试时长为 24 小时.

为了测试 IntHunter 系统对高可信软件的整数溢出漏洞发掘的有效性,我们选取了 3 个高可信目标软件并升级至最新版本,进行实际测试:

(1) 微软公司开发的高可信服务器操作系统 Windows 2003 中重要的 WINS 服务程序.该服务可以解决计算机名称与 IP 地址的对应问题.WINS 客户机之间进行通信时,可通过 WINS 服务器的解析功能获得对方的 IP 地址,我们将该服务程序更新到最新版本后(即打上 MS04-045 和 MS08-034 补丁)待测.

(2) 微软公司开发的 Windows 2000 Server 中重要的 WINS 服务程序.我们也将其更新到最新版本待测.

(3) 百度公司开发的高可信即时通讯软件 Baidu Hi 2.0 Beta1.该软件目前被百度公司用作百度大型电子商务平台“百度有啊”的安全即时通讯工具.

表 1 显示了 IntHunter 对 3 个目标应用进行自动高危整数溢出错误发掘的性能数据,每个目标软件的路径搜索与整数溢出发掘时间均为 24 小时.

Table 1 Performance results of benchmarks (IoF means integer overflow)

表 1 测试程序集的性能结果(IoF 表示整数溢出)

	Win2K3 WINS	Win2K WINS		Baidu Hi
# New critical IoF bugs	1 (with heap overflow)	1 (with heap overflow)+1 (with DoS)		1 (with stack overflow)
# Generated test cases	1 788	1 550		2 192
Divergence rate (%)	2.5	2		5.3
Test case No. (when IoF is found)	52	33	56	89
# Insts through proc.	730 568 458	519 583 548		1 077 634 800
# Tainted insts	1 414 833	812 577		3 858 517

在对这些程序的测试中,IntHunter 系统为这 3 个应用分别产生了 1 788,1 550 和 2 192 个测试用例,共发现了 4 个新的高危整数溢出错误,其中,3 个严重错误可能导致远程攻击并获取系统控制权,另外 1 个整数溢出将可能引发大量内存被分配的 DoS 攻击.对微软公司两个不同 Windows 下的 WINS 服务进行自动化测试后,IntHunter 自动发现并确认了 2 个伴随高危堆溢出的整数溢出错误以及 1 个可引发 WINS 拒绝服务的整数溢出错误(漏洞检索号分别为 CVE-2009-1923 及 CVE-2009-1924).对 Baidu Hi 2.0 Beta 进行自动化测试后,IntHunter 自动发现并确认 1 处因整数溢出引起的可被远程攻击并执行任意代码的栈缓冲区溢出错误(漏洞号检索号为 CVE-2008- 6444).

表中测试用例失效率项用于反映IntHunter系统对目标软件的测试效率.测试用例失效指系统预计测试用例 Ψ_i可以指导程序抵达状态迁移指令t_i并向预订方向迁移.然而,由于实际运行过程和测试追踪过程中的一些不确定因素(如多线程竞争、求解器求解失败、路径约束不精确等),使得执行路径无法抵达t_i或无法向预订方向迁移,即测试用例 Ψ_i对程序路径执行指导失效.测试用例失效率计算公式为

失效测试用例数量/总测试用例数量×100%.

实际上,任何测试用例的失效都只有在程序追踪并运行到指定指令时才能判断出来.因此,高失效率将直接导致测试系统具有极低的可用性.从表 1 我们可以看出,3 个大型目标软件的测试用例失效率均在 6%以下,表明具有较低失效率的 IntHunter 系统可以准确、有效地对目标软件进行路径搜索.

图 6 给出了被调度执行的前 100 个测试用例对目标软件的代码覆盖情况,其中,代码覆盖曲线是使用污点指令基本块的数量增加情况来表现的.图 6(a)为 Windows 2003 Server WINS 服务的前 100 个被调度测试用例的污点基本块覆盖曲线,直线表示测试第 52 个用例时发现一个高危整数溢出错误;图 6(b)为 Windows 2000 Server WINS 服务的前 100 个被调度测试用例的污点基本块覆盖曲线,直线表示分别测试第 33 和 56 个用例时发现高危整数溢出错误;图 6(c)为 Baidu Hi 即时通讯软件的前 100 个被调度测试用例的污点基本块覆盖曲线,直线表示测试第 89 个用例时发现一个高危整数溢出错误.

尽管污点指令基本块数量只有 1 000~2 000,然而由于污点指令数仅占整个进程执行过指令数的 1%以下,实际执行过程中,IntHunter 系统对全程序空间的代码覆盖率仍旧相当高.图 6 中的竖线表示在该测试用例中发现了高危整数溢出错误,可以看出,在面向高危整数溢出错误的测试用例调度策略下,发现 4 个整数溢出错误的测试用例均被有效地调度,并且每个错误均在最早调度的前 90 个测试用例内发现.

图 7 展示了Baidu Hi中导致堆栈溢出的CVE-2008-6444 高危整数溢出错误的发生原理以及IntHunter发掘该错误的过程.如图 7 所示,Baidu Hi使用指针P_1和P_2采用如下方法来提取正常报文中的版本信息“1.0”:先从后向前移动P_2,直到遇到第 1 个非空格字符后,后退一个字节,即 P_2' 的位置;然后调用*strchr*(P_1,‘␣’),从前向后寻找“1.0”前面的空格,并将指针迁移至该空格位置(标示为 P_1');最后,(int)($P_2' - P_1'$)为版本信息的长度, P_1' 为源指针向目标地址拷贝版本信息字符串.

当P_2指针扫描到“1.0”后面的空格时,存在状态迁移t_i:*assume*(*P_2≠“␣”),IntHunter系统将产生约束条件 ¬(*P_2≠“␣”)∧ *PathCond*($P_{t_{i-1}}$) ,通过可满足求解后,产生用“\1”替换“␣”的测试用例,使指针 P_2' 停留在原地.同理,当程序执行*strchr*(P_1,‘␣’)时,IntHunter系统也产生一个字符“\1”替换原报文“*cm*”后面的“␣”,使得strchr函数将P_1指针扫描到字符“*R*”之后,即图 7 中的错误报文.在这样的输入报文控制下,(int)($P_2' - P_1'$)发生整数下溢,使得

Baidu Hi的长度检查条件(int)($P_2'-P_1'$)<MAX_*LEN*被旁路,进入到strncpy函数中并以(uint)(int)($P_2'-P_1'$)为长度进行数据拷贝.由于(uint)(int)($P_2'-P_1'$)是一个巨大的整数,将导致堆栈拷贝溢出,IntHunter捕捉到这个位置的程序异常并发现执行路径中存在高危整数溢出错误,定位并报告错误.

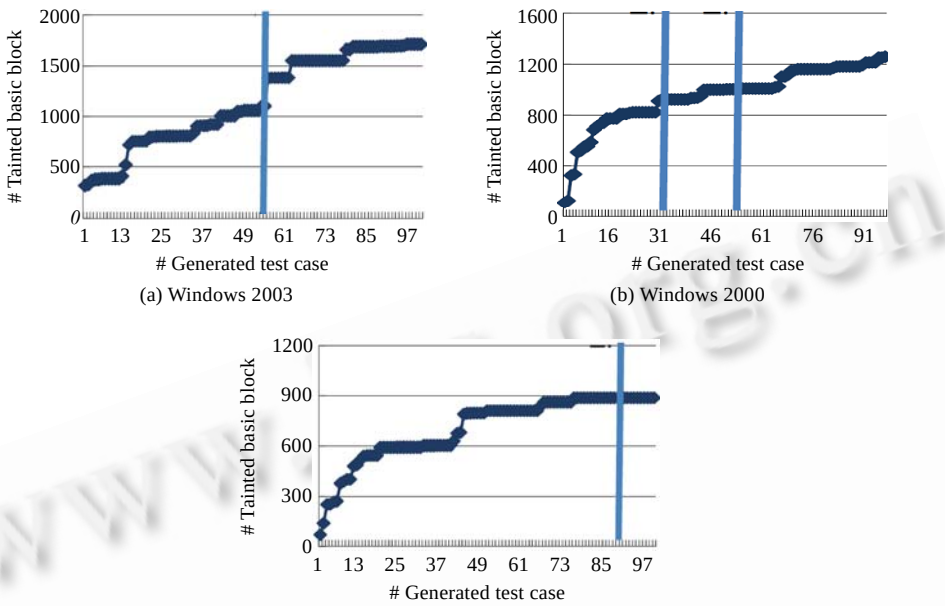


Fig.6 Code coverage of each tested program, the bars represent multiple distinct crashes found early in the search progress

图 6 被测程序的代码覆盖率,图中竖线表示在搜索过程中发现程序崩溃的位置

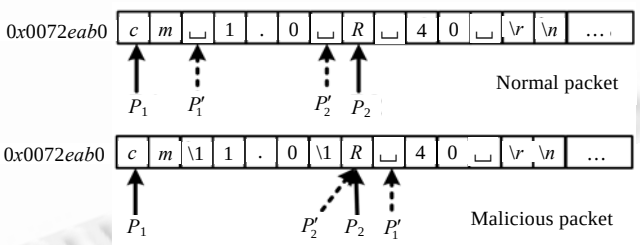


Fig.7 Baidu Hi CVE-2008-6444 stack overflow caused by an integer overflow bug

图 7 Baidu Hi CVE-2008-6444 导致堆栈溢出的整数溢出错误

图 8 给出了 Windows 2000 中可引发高危整数溢出错误的测试用例.该高危整数溢出伴随着堆溢出错误,图 3 中的示例代码就抽象自这段发生高危整数溢出错误的代码.WINS 输入报文中的一个数据域控制了循环的出口条件,并且该数据域经过了严格的边界检查,理想情况下,该报文域的约束条件($len*3*8+4<0x20$)∧ $PathCond(P_{i_{i-1}})$ 使得*len*只能取 0 或 1.然而,IntHunter系统在 2^{32}整数空间中找到了能够满足($len*3*8+4<0x20$)∧ $PathCond(P_{i_{i-1}})$ 的全部其他非程序员预知的 8 个整数:0*xeaaaaaab*,0*xcaaaaaab*,0*xaaaaaaab*,0*x*8*aaaaaab*,0*x*6*aaaaaab*,0*x*4*aaaaaab*,0*x*2*aaaaaab* 和 0*x*0*aaaaaab*.人工方式或随机 Fuzz 测试都难于发现该错误,这也是该错误躲过了微软公司高强度随机黑盒测试的原因.

图 9 给出了Windows 2003 的WINS服务中,高危整数溢出错误发生时的符号化污点数据依赖关系.如图 9 所示,该错误是因为符号化污点 T_{0x222} 可发生整数下溢产生负数,从而进入到程序员没有预料到的路径分支,进行

数据拷贝,进而发生堆溢出错误.

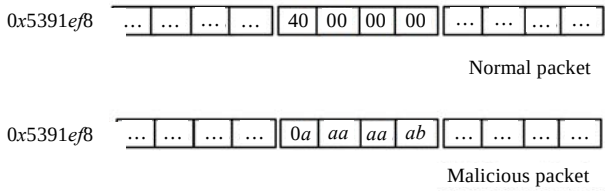


Fig.8 A new integer overflow vulnerability which can cause heap overflow in the latest WINS service of Windows 2000 Server

图 8 Windows 2000 Server 最新版本中,WINS 服务中的一次引发堆溢出的高危整数溢出错误

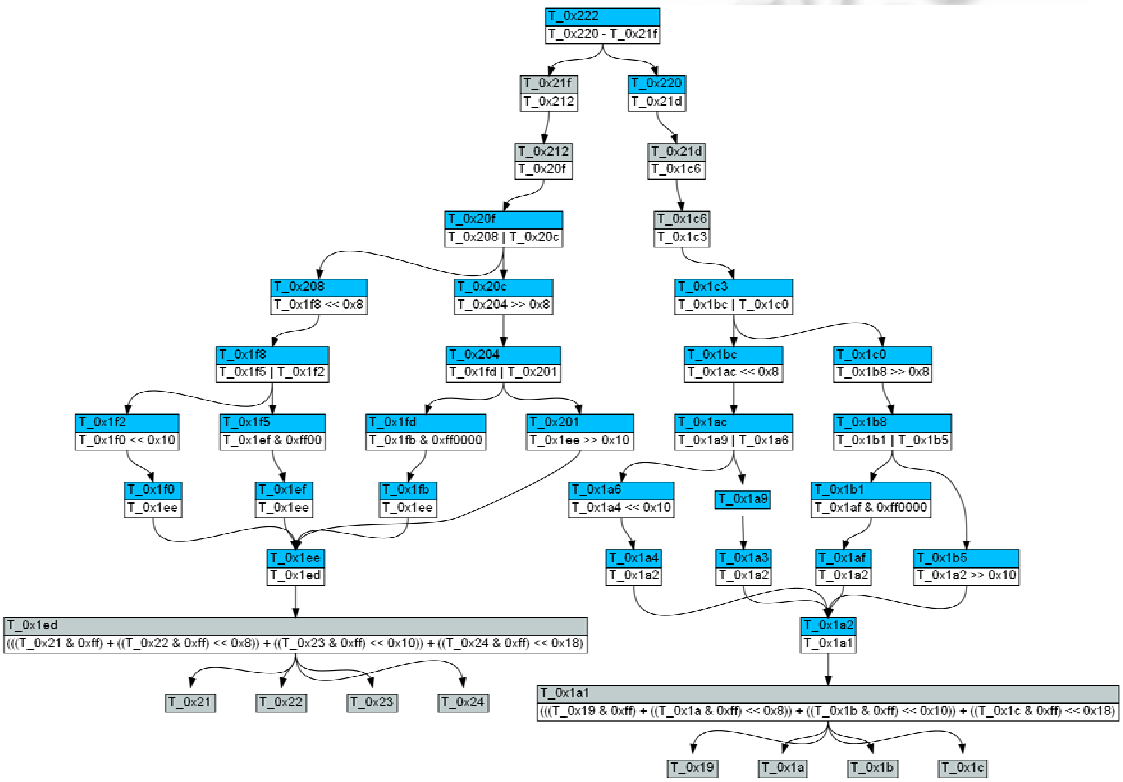


Fig.9 Symbolic taints def-use chain when a new integer overflow vulnerability, causing heap overflow, is met in the latest WINS service of Windows 2003 Server

图 9 按照高可信度标准开发的 Windows 2003 Server WINS 服务中,一处高危整数溢出错误发生时的符号化污点数据依赖关系图

值得一提的是,图 9 所示的数据依赖关系图实际上抽取自一条跨越了 22 个函数、数万条指令的执行路径,可见人工分析该错误十分困难.而 IntHunter 系统无需任何人工干预,自动生成了该高危整数溢出错误的污点数据依赖分析图,通过该依赖图,程序员可以很容易地定位错误发生的原因.

综上所述,我们根据 DAIDT 设计并实现的高危整数溢出发掘系统 IntHunter,在对以上 3 个高可信软件的实际测试中表现出了较高的对该类错误的挖掘效率,同时,成功地自动定位并报告了这 3 款软件中的 4 个未知高危整数溢出错误,充分验证了 DAIDT 理论的可行性、可用性与高效性.

4 相关工作

4.1 源码级整数溢出检测与保护技术

整数溢出错误的发掘研究主要集中在对整数变量不正常使用的检测与防护两方面.

在源码级,采用污点分析与已知的整数溢出模式进行匹配以检测整数变量的使用错误,可有效发现部分常见的整数溢出错误[7,8].当前,主要研究工作有:Ceesay等人通过一个基于Cqual[9]的静态工具来追踪不可信数据;Ashcraft等人采用基于程序员编写的编译扩展来追踪不可信数据,并在追踪过程中检测程序错误[7].

UQBTng[10]是一个在二进制级自动发掘整数溢出的工具,UQBTng首先将二进制代码逆向为C代码,而后在内存分配函数前插入断言,再通过CBMC边界模型约束检查器校验程序属性,并从中发现错误.然而,逆向编译是一个尚未完全解决的问题,UQBTng的有效性受限于该技术;同时,由于UQBTng最终仍旧采用源码形式进行程序检测,所以我们还是将其归于源码级整数溢出挖掘工具.

防护方面的研究有GCC编译器-ftrapv编译选项,通过在算术指令操作前插入溢出检查代码,捕捉运行时过程产生的整数上溢出与下溢出,并及时抛出异常.同样的类似工作有:RICH[11]和GNU多精度运算库GMP[12]等.

总体上,由于源码级技术主要基于静态分析,而整数溢出通常与程序动态的执行路径相关度较大,所以静态分析具有较低的精确性并产生大量误报.

4.2 二进制级动态挖掘技术

以Valgraind[13]/Memcheck[14]和Aftersight[15]等系统为代表的污点追踪与传播技术,可以在不需要任何源码的情况下,运行时检测程序执行路径中是否存在如下错误:对未初始化内存的引用、悬挂指针和内存泄漏等.然而,这些技术只能检测当前执行路径上确实已经存在的错误,无法主动搜索并执行程序中的其他可行执行路径.Flayer[16]和PathExpander[17]等系统为弥补这一缺陷,通过追踪受污点控制的条件跳转指令,并以人工(flayer)或硬件支持(PathExpander)的方式强制修改程序跳转方向,使程序向另一分支执行,以获得较高的路径覆盖率并发现更多错误.然而,这种通过强制改变跳转方向的方法所引发的程序异常,可能在实际执行过程中根本不会发生,而会产生大量的假警报.此外,由于路径修改是强行的,这类系统不能给出引发错误的原因,无法为错误调试提供支持.

SAGE[18]系统是一个二进制级别的自动测试用例生成系统.它采用iDNA系统追踪程序轨迹并记录日志,然后符号化解释执行日志中的x86 指令序列,当遇到与输入相关的分支跳转语句时,记录约束条件,并通过求解器生成指导跳转方向的测试用例.然而,SAGE存在以下的问题:

(1) 离线解释执行丢失了大量有用的运行时信息;

(2) 为了性能而具体化所有的非线性操作指令(如乘法、除法和位运算等),这会导致精度损失.

这些问题致使SAGE系统的测试用例失效率高达60%[18],从而产生较低的代码覆盖和耗费大量无用的符号化解释执行时间.此外,较低的符号化表示精度使得SAGE系统无法有效地应用于高危整数溢出错误发掘.

IntScope[19]是一个基于符号化执行的二进制级整数溢出错误挖掘系统.该系统直接符号化执行二进制x86代码,并从中抽取与输入相关的约束,最后检查与输入数据相关指令是否可能产生整数溢出.若可能产生整数溢出,则用经验性模板寻找该路径上是否已经存在用于防御整数溢出的条件,若存在则忽略该整数溢出,否则报告错误.与我们的IntHunter系统不同,IntScope采用符号化模拟执行,存在精度不高等问题,很难完全准确地模拟真正的运行时环境,这使得IntScope在对程序进行整数溢出错误挖掘过程中容易漏报或误报[19].而我们的IntHunter系统基于动态虚拟化环境,精确完成每条指令,因此,IntHunter系统可以构建位精确的路径约束条件以辅助高危整数溢出错误的精确挖掘.此外,IntScope系统发现整数溢出后,难以确定该整数溢出是否导致程序错误,这是因为程序员可能通过一定的后置路径条件避免该整数溢出产生错误.由于应用程序存在大量的整数操作,如果不能准确判断是否产生错误,则必然导致大量的误报,从而使得系统具有较低的可用性.而我们的IntHunter系统基于真实的运行时系统,可以准确地捕获程序是否因为整数溢出而产生真实程序异常,因此,IntHunter系统可以无误报地自动捕捉并确认高危整数溢出错误.最后,由于解释执行的特性的限制,IntScope

无法准确地处理跨函数库边界甚至跨语言边界调用的大型应用,这使得IntScope的应用范围受限;而我们的IntHunter系统可以直接在全系统空间内监视程序进程的行为,完全透明化大型应用的语言边界和库边界.

5 总 结

本文提出了一种直接在二进制级别对任意高可信软件进行高危整数溢出错误发掘的自动化动态测试方法DAIDT.该方法不需要程序员编写任何测试代码与设计任何测试用例,能够在没有任何源码甚至是符号表的情况下,完全自动地直接对目标软件进行无重复的、精确的、全面的路径覆盖测试,从而有效发掘并准确定位可能导致严重安全事故的高危整数溢出错误.此外,我们提出的精确的高危整数溢出错误确认和定位技术可以保证零误报,并为每个高危整数溢出错误自动生成相应的测试用例,以辅助开发人员或测试人员重现错误.同时,我们从理论上证明了该方法的正确性与可行性.

为了验证DAIDT的有效性,我们实现了原型系统IntHunter,并对微软公司按照高可信度计算标准开发的最新版本的Windows 2003 Server和Windows 2000 Server的WINS服务程序以及百度公司即时通讯软件Baidu Hi分别进行了24小时测试,共发现了4个高危整数溢出错误,其中有3个可能导致任意代码执行.实验结果有力地证明了我们提出的自动化在线高危整数溢出错误发掘技术的有效性.

References:

[1] Chen HW, Wang J, Dong W. High confidence software engineering technologies. Acta Electronica Sinica, 2003,31(B12): 1934−1938 (in Chinese with English abstract).

[2] Christey S, Martin RA. Vulnerability Type Distributions in CVE. The MITRE Corporation, 2007. 1−38.

[3] Gries D. The Science of Programming. New York: Springer-Verlag, 1981. 107−163.

[4] Ball T, Ragamani SK. Generating abstract explanations of spurious counterexamples in C programs. In: Proc. of the MSR-TR-2002-09. Redmond: Microsoft Corporation, 2002. http://research.microsoft.com/research/pubs/view.aspx-msr_tr_id=MSR-TR-2002-09

[5] Cifuentes C, Sendall S. Specifying the semantics of machine instructions. In: Proc. of the Int'l Workshop on Program Comprehension. Washington: IEEE Computer Society Press, 1998. 126−133.

[6] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Damm W, ed. Proc. of the Computer Aided Verification. Berlin: Lecture Notes in Computer Science, 2007. 524−536.

[7] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society Press, 2002. 143−159.

[8] Ceesay EN, Zhou J, Gertz M, Levitt K, Bishop M. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. In: Büschkes R, Laskov P, eds. Proc. of the Detection of Intrusions and Malware & Vulnerability Assessment. Berlin, Heidelberg: Springer-Verlag, 2006. 1−16.

[9] Zhang X, Edwards A, Jaeger T. Using CQUAL for static analysis of authorization hook placement. In: Dan B, ed. Proc. of the Usenix Security. San Francisco: USENIX Association, 2002. 33−48.

[10] Wojtczuk R. UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries. In: Proc. of the 22nd Chaos Communication Congress. Berlin: Chaos Computer Club, 2005.

[11] Brumley D, Song DXD, Chiueh TC, Johnson R, Lin HJ. RICH: Automatically protecting against integer-based vulnerabilities. In: William A, ed. Proc. of the 14th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2007.

[12] Howard M. Lessons learned from the animated cursor security bug. 2007. http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx

[13] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, 42(6):89−100.

[14] Nethercote N, Seward J. How to shadow every byte of memory used by a program. In: Proc. of the ACM/Usenix Int'l Conf. on Virtual Execution Environments. New York: ACM Press, 2007. 65−74.

[15] Chow J, Garfinkel T, Chen PM. Decoupling dynamic program analysis from execution in virtual environments. In: Proc. of the 2008 Usenix Annual Technical Conf. Boston: USENIX Association, 2008. 1−14.

[16] Drewry W, Ormandy T. Flayer: Exposing application internals. In: Proc. of the 1st USENIX Workshop on Offensive Technologies. Boston: USENIX Association, 2007. 1–9.

[17] Lu S, Zhou P, Liu W, Zhou YY, Torrellas J. PathExpander: Architectural support for increasing the path coverage of dynamic bug detection. In: Proc. of the 39th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Washington: IEEE Computer Society Press, 2006. 38–52.

[18] Godefroid P, Levin MY, Molnar DA. Automated Whitebox fuzz testing. In: Proc. of the 2008 Network and Distributed System Security Symp. San Diego: ISOC, 2008.

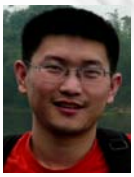
[19] Wang TL, Wei T, Lin ZQ, Zou W. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proc. of the 2009 Network and Distributed System Security Symp. San Diego: ISOC, 2009.

附中文参考文献:

[1] 陈火旺,王戟,董威.高可信软件工程技术.电子学报,2003,31(B12):1934–1938.

卢锡城(1946－),男,江苏靖江人,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能计算,并行与分布处理,先进网络技术.

卢凯(1983－),男,博士,教授,硕士生导师,主要研究领域为高性能计算,操作系统设计.



李根(1982－),男,博士生,主要研究领域为系统安全,操作系统设计.

张英(1981－),女,博士,讲师,主要研究领域为高性能计算,编译技术.