2.1 函数调用索引与返回地址索引

在解析 Trace 文件过程中,如遇到函数调用(call)指令,将其压入函数调用栈中,并且将函数调用深度增 1.对于每个线程,都维护一个函数调用栈.接下来,遇到每条指令都进行比较,判断其是否是函数调用指令的下一条指令(即返回后的下一条指令):若是,则认为它是该函数调用指令的返回地址,并将函数调用深度进行调整.如果该返回地址与函数调用栈中的栈顶元素相匹配,则函数调用深度减 1;否则,需减去它到栈顶的距离,它到栈顶的元素全部出栈(如图 2 所示).把函数调用指令及其返回指令在 Trace 文件的偏移、函数调用深度、线程号均记入函数调用索引文件中.

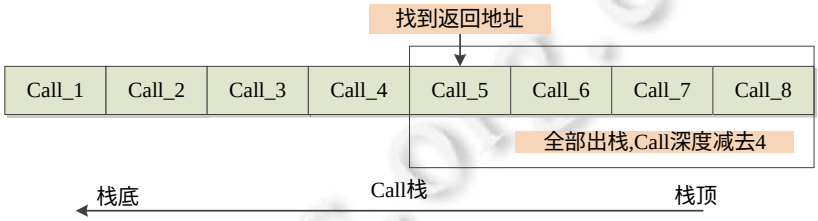

Fig.2 Operation of call stack

图 2 调用栈操作

返回地址索引的主要作用是为了根据返回地址反向查找其函数调用指令的地址.当函数调用索引文件生成完成后,通过读取函数调用索引文件,将返回指令在踪迹文件中的偏移作为关键字进行排序,然后写入返回地址索引中.

由于函数调用索引与返回地址索引中的记录都是有序的,且每个索引记录的大小都是固定的,因此可使用二分查找算法(binary search)来查询,时间复杂度为 $\log_2(n)$,n 为索引记录的个数.

结合函数调用索引和返回地址索引,可快速地确定函数边界,如果函数包含在用户定义的白名单中,则不需要对其作污点分析,可直接根据函数边界跳过该函数过程,因而可极大地提高污点分析的效率.白名单是一个函数列表,具有两种表示形式:一种是模块名和偏移:{Module:Offset},另一种是函数名称.这两种方式都可唯一标识函数,函数名称便于用户设定,但需要符号信息的支持.

算法 1 中确定函数边界的方法是:给定指令的偏移,先判断其是否为函数调用指令:若是,则直接读取函数调用索引得到其返回地址所在偏移;否则,先往下查找返回指令(ret 或 retn),找到后在返回地址索引中查找对应的函数调用所在偏移.如果未找到,则往上寻找函数调用指令,直到找到函数调用指令为止.注意:在往上查找函数调用指令时,如遇到返回指令,则需查找到对应的函数调用指令,并直接跳过该函数继续向上查找.

算法 **1**. 确定函数边界的算法.

输入:目标指令所在偏移;

输出:函数的边界(即调用地址和返回地址).

```
begin
1   re=DecodeInsn(addr)
2   if IsCallInsn(re) then
3      FindReturnByCallIndex(addr)
4   else
5      while (addr<MAX_ADDR) do
6         re=DecodeInsn(addr)
7         if IsReturnInsn(re) then
8            success=FindCallByRetIndex(addr)
9         endif
10     endwhile
```

```
    if success!=TRUE then
      while (addr>MIN_ADDR) do
        re=DeocdeInsn(addr)
        if IsCallInsn(re) then
          FindReturnByCallIndex(addr)
        endif
      endwhile
    endif
  endif
end
```

2.2 内存索引与指令地址索引

内存索引可快速定位访问指定内存地址的指令记录,内存索引的生成方法是:将内存地址空间分成页,每页的大小为 *PAGE_SIZE*(默认大小为 32KB).页内的所有地址对应同一个队列,队列中的每个元素为在踪迹文件中偏移的区间(interval),表示该内存页中的某些地址落入这个范围中.当某条指令所在文件偏移与其所在队列的最大文件偏移相差超过阈值 *OFFFSET_THRESHOLD*(默认为 1024B)时,此队列添加一个新的区间,其上界与下界相等,都为该指令在 Trace 文件中的偏移.若未超过此阈值,则修改队列中最后一个区间的上界即可.

查询内存索引的过程:先对所需查找的内存地址计算页号,使用页号查找内存索引,查找到对应的分页后取出对应的地址区间队列,遍历所有地址区间内的指令记录.如果指令访问内存的地址与要查找的内存具有交集,则返回该指令的位置.

其中,*PAGE_SIZE* 与 *OFFSET_THRESHOLD* 的值是根据实际程序的测试得出的经验值,以 CVE-2011-0558 漏洞的分析过程作为选择经验值的样本,跟踪所得踪迹文件的大小是 7.7GB.从表 1 的数据可看出:若 *PAGE_SIZE* 与 *OFFSET_THRESHOLD* 的值越小,则生成的索引文件越大,但分析所用的时间也越少;反之,生成的索引文件越小,但分析所用时间越多.因此,当 *PAGE_SIZE* 为 32KB,*OFFSET_THRESHOLD* 为 1024B 时,污点分析的效率最高,且生成索引的大小也在可接受范围.

Table 1 Experiment of choosing empirical value

表 **1** 经验值选取实验

#PS/KB	*#OT*/KB	*#R*/MB	*#Size*/MB	*#Time*/s
64×32	10	8	12	54
64×32	10	4	12	52
64×32	10	1	12	75
64	1	8	176	57
32	1	8	220	20

在表 1 中,*#PS* 表示 *PAGE_SIZE*,*#OT* 表示 *OFFSET_THREASHOLD*,*#R* 表示调用内存索引间隔的大小,*#Size* 表示生成索引的大小,*#Time* 表示污点传播所使用的时间.内存索引的查询算法见算法 2.

算法 **2**. 内存索引查询算法.

输入:所需查找目标内存地址;

输出:访问目标内存地址的所有指令位置.

```
begin
1  IndexQuery=CaculateIndex(QueryMemoryAddr)
2  RangeQueue=GetRangeQueue(IndexQuery)
3  for range in RangeQueue do
4    (LowAddr,HighAddr)=range
5    for addr in [LowAddr,HighAddr] do
```

```
6      MemoryRange=ReadInsnRecord(addr)
7        if Intersection(Memoryrange,QueryMemoryAddr)!=nil then
8          AddList(addr)
9        endif
10     endfor
11 endfor
end
```

2.3 黑白名单机制

本文设计了较灵活的用户配置机制,其中,通过黑名单机制,可动态地在某个程序点自定义地进行污点的引入或消除.通过白名单机制,可跳过一些与污点无关的函数,以提高污点分析的效率.对于当前有些程序,污点引入与文件读取的关联性极其隐蔽.如 LibTiff 库,其对图片文件的处理流程为:首先,将图片全部载入其内存中;然后,使用自写的读取函数(如 *TIFFGetField*)对内存中的图片数据进行解码.由于图片自身的数据大多经过压缩,解码过程常将少于 4 字节的机器码解码为一个整数,若按照以往的方法,只对 *ReadFile* 函数下断,则无法取得预期的效果.本文为解决该问题,设计了一个方便的接口,用户可根据该接口提供参数,以供在污点分析过程中机动地引入或消除污点数据.用户提供的参数包括:需下断的函数名或函数地址、需变动污点的地址、需变动污点的大小、引入污点或消除污点.

具有上述机制后,对 LibTiff 库的处理可直接对其自写的读取函数设置断点,并将其返回值作为新的污点引入.黑名单的定义完全由用户提供,需用户对分析的目标程序有初步的理解,用户需通过逆向分析来确定需动态改动污点信息的位置,自动化地确定该位置将作为下一步的研究工作.

对于有些函数(如 *CreateFile*,*GetProcessId* 等),其执行过程并不会影响污点数据的变动,因此无需步入函数跟踪污点数据,为此类函数建立一个白名单(white list),当调用函数位于白名单中时,可通过这索引直接跳过即可.检测函数是否在白名单中的方法为:对于系统库函数,可通过符号名直接对其进行匹配;而对于无符号信息的用户自定义函数,则可通过库名与偏移的形式进行比较.白名单可通过两种方法进行添加.

1) 通过库函数说明文档(MSDN)对函数的描述添加;
2) 在测试实际程序过程中,根据测试中获得的经验添加.

白名单仅为提高污点分析的效率,白名单中函数越多,表明在污点分析过程中可跳过的函数也越多,污点分析过程的效率也越高.

2.4 JIT翻译问题及解决方法

在某些支持 JIT(just in time)编译执行的程序(如 Flash)中,污点数据可能直接被复制为指令的操作数,成为指令的一部分.在该情况下,代码与数据的概念冲突,若按照传统的污点分析方法,污点信息必然会发生丢失,本文提出一种基于该问题导致污点信息丢失的解决方法.某些程序在执行过程中,动态地生成代码并将这些代码复制到缓存区中,然后跳转到缓存区中去执行,这种执行方式称为 JIT 翻译执行.

JIT 翻译执行可能会造成污点丢失的情况,因为在生成代码时,程序可能将污点数据直接以机器字节码的形式复制到缓存区中.如图 3 所示,污点数据被转化为该条指令中的立即数,使用传统的污点分析算法无法感知这种转化,因此必然会导致污点的丢失.

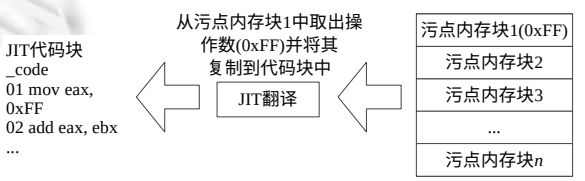

Fig.3 Operation of JIT translation

图 3 JIT 翻译操作

本文借助生成的指令地址索引来解决此问题,当污点数据被转化为指令中的一部分时,必然具备如下特征:

$$S_{TAINT} \cap S_{EIP} \neq \varnothing,$$

其中,S_{TAINT} 表示当前所有污点数据内存地址的集合,S_{EIP} 表示所有指令地址的集合.该语句是检查污点数据地址集与指令地址集是否发生重叠,在污点传播过程中,若检测到此类重叠,则需继续将污点传播到正确的位置,才能保证污点分析过程的顺利进行.

如果对每条指令作污点传播时都检测指令地址与当前污点地址是否发生重叠,无疑会极大增加运行的时间.指令地址索引的引入,则可有效提高污点分析算法的运行效率.

支持跳转的污点分析过程见算法 3.

算法 **3**. 支持跳转的污点分析算法.

输入:当前指令位置;

输出:污点分析结果.

```
begin
1  while (CurrentLocation<MAX_LOC) do
2     if IsAllRegisterTaintEmpty() then
3        loc1=FindMemoryAccessByIndex(TaintSet)
4        loc2=FindEIPOverlapsByIndex(TaintSet)
5        loctojump=Min(loc1,loc2)
6        CurrentLocation=loctojump
7        continue
8     endif
9     TaintAnalysis(CurrentLocation)
10    CurrentLocation++
11 endwhile
end
```

综上所述,提高污点传播的效率需结合内存索引与指令地址索引,在污点传播过程中,每执行一条指令时,检测当前程序状态下是否所有寄存器的污点都为空:若是,则根据内存索引可查询下一次访问污点内存的指令位置 loc_1.

如果需考虑 JIT 翻译执行的情况,则需查询指令地址索引,检测是否存在上述污点数据与指令地址重叠的情况,并找到发生重叠的最近的指令位置 loc_2.此时,可将当前污点传播位置跳到位置 $loc=\text{Min}(loc_1,loc_2)$.

3 系统实现

本文以上述基于执行踪迹离线索引的、支持污点标签、以字节为污点传播粒度的污点分析方法为理论依据,使用 2 万余行 C++代码实现了一个污点分析工具,其中,指令跟踪模块 4 千余行,踪迹文件分析与索引生成模块 1 万余行,污点分析模块 6 千余行.

如图 4 所示,该系统包括 4 大模块.

1) 指令跟踪模块:监控程序执行,并实时记录污点分析所需的上下文信息,将其写入踪迹文件中;
2) 踪迹文件分析与索引生成模块:用来分析生成的踪迹文件,并生成函数调用索引、返回地址索引、内存索引和指令地址索引;
3) 污点分析模块:在踪迹文件上进行离线的污点分析;
4) 漏洞检测模块:在污点分析完成后,根据特定漏洞模式来检测漏洞,并生成报告.

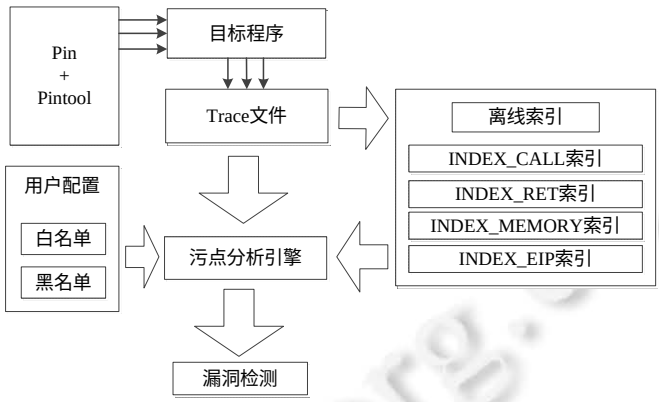

Fig.4 Structure of our system

图 4 系统结构图

3.1 指令跟踪模块

该模块使用动态插装工具来对实际程序进行插装,监控其执行过程.本文的实现是基于英特尔公司的二进制插装工具 Pin[2−6],根据其提供的用户接口,为污点分析实现了一个相应的插装工具.指令跟踪模块包括 3 部分.

(1) 增量记录寄存器上下文

踪迹文件中,指令记录占绝大多数,其中以寄存器的值所占空间最大.为减少踪迹文件所占的磁盘空间,本文使用增量式的记录方法,对每条指令只记录当前指令引用或修改的寄存器的值.与记录所有寄存器的值的方法相比,该方法所占存储空间约为前者的 1/32.通过实现相应算法,向前或向后遍历,即可方便地推算出某个程序状态所有寄存器的值.

(2) 外部输入与内存映射

通常情况下,将外部输入看做是污点来源,例如文件中的某些字段、网络包中的某些字节.程序将外部输入读进内存,并引入污点数据,因此需映射外部输入到内存$\{I\mapsto M\}$,以获得污点数据的内存地址.通过劫持诸如读取文件、接收网络数据包及读取注册表信息等系统函数,并根据其返回参数,可生成外部输入与内存地址之间的映射.表 2 归纳了一些较常见的外部输入方式,并说明了其参数与返回值.

Table 2 External input of several functions

表 **2** 外部输入函数

函数名	模块名	摘要
ReadFile	Kernel32	第 2 个参数返回地址,由第 4 个参数返回读取大小
MapViewOfFile	Kernel32	返回映射的地址,由第 5 个参数指定映射内存的大小,如果为 0,表示映射从 offset 到文件最后
MapViewOfFileEx	Kernel32	返回映射的地址,由第 5 个参数指定映射内存的大小,如果为 0,表示映射从 offset 到文件最后
recv	Ws2_32	由第 2 个参数返回接收的地址,返回值表示长度
WSARecv	Ws2_32	由第 2 个参数返回接收的地址,第 4 个参数表示长度
WSARecvFrom	Ws2_32	由第 2 个参数返回接收的地址,第 4 个参数表示长度
recvfrom	Ws2_32	由第 2 个参数返回接收的地址,返回值表示长度
RegQueryValue	Advapi32	由第 3 个参数返回地址,第 4 个参数返回长度
RegQueryValueEx	Advapi32	由第 5 个参数返回地址,第 6 个参数返回长度
RegGetValue	Advapi32	由第 6 个参数返回地址,第 5 个参数返回长度
RegQueryMultipleValues	Advapi32	由第 4 个参数返回地址,第 5 个参数返回长度

(3) 踪迹文件生成

踪迹文件中包括两类记录:① 模块记录,记录模块加载时的信息;② 指令记录,记录当前指令的信息,包括

时间戳、线程号、指令码、内存访问地址及增量寄存器的上下文信息.为优化程序在动态插装下的执行效率,应尽可能地提高 I/O 效率,本文使用了延迟写的记录方式,即:每次指令执行前先将其存入缓存(cache)中,待缓存填满后再写入踪迹文件中.

3.2 踪迹文件分析与索引生成模块

踪迹文件生成后,需对其进行离线分析,根据记录的文件格式进行解析.在此过程中,同时生成 4 种索引:函数调用索引、返回地址索引、内存索引和指令地址索引,这 4 种索引的生成与查询方法已在上一节中描述.

踪迹文件分析完成后,可提供如下功能:获取模块信息,取得所有模块的模块名、加载地址;获取线程信息,取得所有线程的线程号、线程起始地址,根据在踪迹文件中的所在偏移取得指令记录;支持符号信息,可根据微软提供的符号信息,将对应的地址转化为符号名.

3.3 污点分析模块

指定外部输入中的某些字节,通过外部输入与内存的映射,可得到其在内存中的位置与大小,由此通过 SetTaint 原语引入污点源,由引入污点的位置开始进行污点传播过程.

污点元可以是寄存器或内存,为支持多线程的程序,寄存器必须是线程相关的.为统一管理污点标签,我们实现了统一的污点标签管理方法,把所有污点元的污点标签统一进行管理,内存使用内存地址作为标识,寄存器的标识使用高于 2G 空间的内存地址来表示.将寄存器 ID 与相应线程的 ID 经哈希运算得到的值,作为寄存器的标识来操作对应的污点标签.

- 对于内存:$Mem \mapsto Taint$;
- 对于寄存器:$\mathbf{H}(RegID, Tid) \mapsto Taint$.

污点传播过程直接在汇编指令上进行,需考虑指令的语义和副作用.汇编指令大致可归纳为 3 种类型的操作:1) 数据传输指令,如 MOV,PUSH,XCHG 操作;2) 运算指令,如 ADD,SHL,OR 操作;3) 跳转指令,如 Jmp,Call 操作.对于污点传播主要考虑前两种类型的指令.

定义通用的污点传播语义如下.

- 数据传输指令:$\dfrac{mov\ dst,\ src}{T_{dst} \leftarrow T_{src}}$;
- 一元运算指令:$\dfrac{Uniop\ src}{T_{src} \leftarrow T_{src}}$;
- 二元运算指令:$\dfrac{Binop\ dst,\ src}{T_{dst} \leftarrow UNION(T_{dst}, T_{src})}$.

对于大部分指令,都可使用通用的污点传播方式,有些指令则需作特殊处理,如移位操作 SHL,SHR,SAR 和浮点栈操作.

- 移位操作.由于本文所实现的污点分析方法是基于字节粒度的,对于移位操作,如果移动的位数大于 8 位,则进行污点传播.例如,一个 4 字节的整数向左移 8 位,将低 3 个字节的污点标签复制到高 3 个字节,并且将最低字节的污点属性消除;
- FPU 浮点寄存器栈.对于浮点寄存器栈也需要作特殊处理,如 FPU 寄存器的栈顶始终为 *ST*(0),对于出栈操作 FSTP,除了将栈顶 *ST*(0)的污点属性传输到目的操作数外,还需将所有栈中元素向上移一个位置.入栈操作 FLD 则相反,需要先将所有栈中元素向下移一个位置;
- 扩展操作.对于类似 MOVZX,MOVSX 的扩展操作,除了将源操作数扩展成 4 个字节的整数外,还需要将相应高字节部分的污点属性都消除;
- 指令的副作用.有些指令具有副作用,如 REP 操作影响寄存器 ECX 的值,大部分运算操作影响标志寄存器 EFLAGS 的值,在污点传播时必须加以仔细考量.

在污点传播过程中,需要进行污点消除以避免污点的非法扩散,这涉及以下几种情况:① 指令清空污点元,

如 XOR x,x 类型的指令;② 污点元被覆盖,如使用数据传输指令,将常数赋值给污点元,或将一块不具有污点属性的内存传递给污点元;③ 指令的副作用,如 CDQ 指令会默认把寄存器 EDX 清空.

表 3 中列出一些污点消除的情况.

Table 3 Several examples of taint elimination

表 3 一些污点消除示例

指令类型	污点消除的条件
XOR, PXOR	源操作数与目地操作数相同
SUB	源操作数与目地操作数相同
CDQ/CWD	无需条件
MOV	源操作数为常数
AND	源操作数为常数,且包含 0 字节
OR	源操作数为常数,且包括 0xFF 字节
FLD1, FLDPI, …	无需条件

为达到分析精度和分析效率的折衷,本系统支持用户自定义配置污点传播的规则,可供用户根据自己的需要来灵活配置的选项有:① 如果内存地址为污点,其存储的值是否也认为是污点数据;② 是否将循环操作的计数作为污点数据;③ 是否考虑标志寄存器的污点属性;④ 当移位操作的移位个数是污点时,结果是否也认为是污点.

本系统支持用户自定义的白名单(white list)与黑名单(black list):可通过白名单的方式,将不需要进行污点传播的函数列出,遇到此类函数时即可直接跳过,以提高污点分析的效率;还可提供一个黑名单,用户可自定义这些函数的污点传播规则,如 *strlen* 函数,如果其参数为污点数据时,可将函数的返回值也置为污点数据.

3.4 漏洞检测模块

污点分析过程完成后,可以确定污点数据从污点源传播到了哪些目标位置.一般来说,污点数据是可控的,因此,当污点数据以参数形式传入某些函数时,则其可能导致漏洞的出现.

本文主要考虑 3 种漏洞模式,分别是内存任意读写、内存分配不当导致的溢出问题以及调用危险函数,这 3 种漏洞模式适用于污点分析方法.对于内存任意读写,只需检测是否将污点数据写入由污点数据表示的内存地址中.这一模式可描述为[*Taint*]←*Taint*.

对于内存分配不当,只需检测分配内存的函数,如 *alloc*,*malloc*,*operator new*,*VirtualAlloc* 等,其参数是否为污点数据.

有些危险函数如 *memcpy*,*strcpy*,*memmove* 等函数也容易导致溢出漏洞的出现,因此也需检查其参数是否为污点数据.此外,还可由用户自定义危险函数,若其参数为污点数据时,则进行检测.

4 实验结果

为证明本文提出的污点分析方法行之有效,本文选取了近年来若干较典型的漏洞作为测试样本集进行测试.选取样本漏洞的依据为:此类样本的漏洞类型多为整数溢出漏洞,且漏洞成因与文件输入直接相关,从理论上可使用污点分析方法进行检测.

在测试前,需对这些漏洞进行分析,使用相应的样本文件作为输入,检测其是否能将污点数据传播到对应的漏洞触发位置,进而由本文所提出的方法进行检测.

为与传统的漏洞检测方法进行比较,采用伯克利大学研发的原型系统 BitBlaze[7−10]中的 TEMU[10−12]作为比较对象,TEMU 中所使用的污点分析方法具有一定代表性,并且开放源代码,但其自身运行于 Linux 操作系统平台,并且与虚拟机 QEMU 结合,为与本文中的方法进行比较,需对其进行如下改动:(1) 使其可兼容于 Windows 系统平台;(2) 将在线的污点分析过程改为离线.由于未对 TEMU 的污点分析的核心代码改动,因此,修改后的系统可如实地反映 TEMU 的分析效率.

实验环境为 CPU:Intel Core i5-3337U 1.8GHz;内存:4GB;操作系统:Windows 7 SP1(64 位).

通过表 4 中的实验数据可知:相比传统的污点分析工具来看,在验证漏洞方面,本文提出的污点分析方法具备较强的漏洞检测能力,这是由于:

1) 本文提出的方法加入的黑名单机制使其在检测漏洞方面更加具有灵活性,使其可检测的漏洞模式包括内存任意读写、内存分配不当、调用危险函数,较 TEMU 更多;例如对 CVE-2010-0304 漏洞,需检测的危险函数为 *tvb_get_nstringz*,该函数第 3 个参数为读取的长度,由网络数据包中指定,为污点数据;第 4 个参数为栈变量.当第 3 个参数大于第 4 个参数时,造成栈溢出;
2) 本文提出的方法解决了 JIT(just in time)翻译执行导致的污点丢失问题,因此可处理使用了 JIT 机制的程序.例如 CVE-2012-0774 漏洞,若不解决 JIT 翻译问题,则会导致污点信息在传播过程中丢失,进而无法检测该类漏洞.

Table 4 Experiment of vulnerabilities detecting

表 **4** 漏洞检测实验

漏洞编号	TEMU 使用的污点分析方法		本文提出的污点分析方法		检测规则
	是否可验证	检测时间(s)	是否可验证	检测时间(s)	
CVE-2009-2347	Y	163	Y	36	*malloc*
CVE-2009-3459	Y	669	Y	127	*malloc*
CVE-2010-0188	Y	146	Y	27	*memcpy*
CVE-2010-0304	–	–	Y	243	*tvb_get_nstringz*
CVE-2010-3333	–	–	Y	155	内存任意读写
CVE-2011-0104	–	–	Y	149	*memcpy*
CVE-2011-0978	–	–	Y	127	内存任意读写
CVE-2012-0003	–	–	Y	350	内存任意读写
CVE-2012-0158	–	–	Y	171	*memcpy*
CVE-2012-4564	Y	210	Y	43	*malloc*
CVE-2012-0774	–	–	Y	156	内存任意读写
CVE-2012-1535	–	–	Y	78	*malloc*

在执行效率方面,本文提出的基于执行踪迹离线索引的污点分析方法更具有明显的优势,平均比传统的污点分析方法快 5 倍以上.

我们将本文所提的方法部署在漏洞挖掘系统平台中,已发现大量软件漏洞,其中发现了金山 WPS、Xnview 等软件中 10 个 0day 漏洞,已提交给国家信息安全漏洞库.出于安全性与厂家协作等考虑,故不在此批露这些漏洞的细节.

4.1 讨 论

二进制代码的污点分析方法是一种近似准确的数据流分析方法,可通过污点分析方法观察到敏感数据流向何处,如果其传入敏感位置时,则可能导致漏洞.然而,由于污点分析方法只关注数据流向而不关心数据的值,导致在有些情形下可能产生污点分析的不确定性,即污点分析方法的盲区.在实际测试中,发现污点分析算法中,盲区主要包含以下 3 种情况.

① 污点数据扩散

考虑以下这种情况:

$$\begin{cases} x = y + z \\ z = x - y \end{cases}.$$

假设 y 为污点数据,那经过第 1 条语句执行后,根据污点传播算法,x 也成为污点数据.同理,当执行完第 2 条语句后,z 也成为污点数据.然而从语义上看,z 的值并未发生变化,永远保持其原有的值,因此,将其认为是污点数据是错误的.类似这种情况,将会引发污点数据的扩散.

② 污点数据丢失

污点信息同样可能会由于语句中的副作用发生丢失,如 *strlen* 函数,当该函数的参数为污点数据时,理论上该函数的返回值的大小也应被认为是污点数据.但是按照传统的污点传播算法,这类污点信息必然会丢失.本文

提出函数黑名单的目的就是想解决这类型的问题,但如果用户内联实现 *strlen* 函数,则无法通过此方法解决.类似地,循环结构中,循环上界与循环次数之间也可被认为是相关的,尽管不少研究者[13,14]提出了对循环结构的识别,尽可能地弥补由循环引起的污点信息丢失,但仍无法适用于全部情况.

③ 污点传播不当

污点传播不当,同样是由于污点分析方法只关注数据流而不关心其值所引起的问题,考虑如下的情况:

$$\begin{cases} x = y \times z \\ y = x \end{cases}.$$

假设 y 为 4 字节整数,其低两个字节为污点数据,当这两条语句执行完成后,很难确定 y 到底含有几个字节的污点数据,因为其结果与 z 的值相关.如果 z 的值为 1,则 y 的污点信息未发生变化;而如果 z 为 0,则 y 应该不再具有污点属性.*fscanf* 函数中也存在这样的问题,以下是其内部的几行代码:

$$\begin{cases} x = SHL(x,4) \\ \ldots \\ x = SHL(x,4) \end{cases}.$$

SHL 为左移指令,表示连续两次将 x 左移 4 位,两次左移达到 8 位,将发生对字节的污点传播.然而本文中实现的方法是以字节为粒度,只对左移 8 位以上才对污点数据进行移位.对这种情况无法感知,因此会发生污点信息传播不当的问题.

5 相关工作

近年来,国内外很多学者都围绕污点分析方法进行了较深入的研究[12−22].Yin 等人[13]提出了基于 QEMU 虚拟机的扩展平台 TEMU,可解决动态分析的困难,便于在其上进行程序分析.TEMU 使用全系统的视角,可分析内核中的活动与多进程间的交互;提供了很好的隔离性与透明性,使恶意程序难以检测到分析环境并影响分析结果,以细粒度的方式进行深度分析.

Schwartz 等人[15]描述了一种基于中间语言上的污点传播策略.

1) 污点引入:实现了一个简单的用户输入源——*get_input* 调用,表示从系统调用、库调用返回的结果,不同的输入源的污点引入规则也不同;
2) 污点检查:污点的状态值用以确定程序的异常行为,通过添加此类规则到操作语义的前提条件中进行检查;
3) 污点传播策略:内存操作包括内存的地址和内存单元两类值,污点传播策略分别追踪内存地址和内存单元的值;
4) 污点消除.如程序函数计算的结果恒为常量时,则需要消除污点.

Enck 等人[16]描述了 TaintDroid,一个基于动态污点传播方法的移动手机的扩展平台,可追踪私密的敏感信息流.其特性包括:(1) 借助虚拟机的解释器提供变量级的追踪,使用解释器提供的变量语义来避免 Intel x86 指令的污点信息扩散.对变量进行修改,并维护其污点标记;(2) 实现应用程序间的消息追踪,可减小进程间通信的成本;(3) 对于系统提供的库,使用函数级的追踪.直接运行库中的代码,并在其返回时修改污点传播策略,需事先可知函数的语义信息;(4) 使用文件级的追踪方法,确保持久信息保持其正确的污点标记.

Kang 等人[17]提出一种动态的污点传播方案 DTA++,包括两个阶段:首先通过诊断,产生不完全污点的分支生成规则,并确定离线分析所需额外的传播;其次,在以后动态污点分析中应用那些规则.其基本原理是:查找不完全污点的控制流的路径,由于确定输入需更多细粒度的信息,使用符号执行和基于路径谓词的方法.在符号执行中,指令执行的踪迹对应包括一系列分支判定的程序路径约束.查找分为两部分:检测谓词Φ确定一个路径子串中是否含有引发错误的隐式流;然后查找Φ中第 1 个路径前缀.

北京大学的王铁磊等人提出了 TaintScope[18]和 IntScope,其中,TaintScope[19]用于定位程序中的校验和,而 IntScope 的主要贡献是利用污点分析方法来检测整数溢出漏洞.其实现方法是:通过反编译方法,将二进制代码

翻译成为静态单赋值形式的中间表示,建立控制流图和调用图,使用深度优先遍历控制流图.对于分支,则使用求解器计算当前路径约束是否满足该分支的条件.

北京航空航天大学的忽朝俭等人[20]提出并深入分析了驱动程序中频繁出现的写污点值到污点地址漏洞模式,提出了一种针对该种漏洞模式的基于反编译、数据流分析和污点分析的检测方案,并实现一款既可分析本地二进制代码,也可分析 C 源代码的原型工具.

上述传统的污点分析方法为本文的研究工作提供了坚实的理论基础和实现细节,但其中大多数污点分析方法不支持浮点指令,执行效率较低,且传播的精度也不够高.本文为解决这些问题,在前人的工作基础上进行更进一步的探索与研究.

6 结束语

本文实现了一种基于执行踪迹离线索引的污点分析方法,支持污点标签,并以字节为粒度进行污点传播.离线索引可跳过与污点数据无关的指令,极大地提高了污点分析的效率.本文首次描述并解决了由 JIT 翻译执行导致的污点丢失问题.利用污点标签,可标识污点来源于何种外部输入及其对应外部输入的哪些字节.以字节为粒度增加了污点传播的精度,保证污点信息以较合理、准确的方式从源操作数传递到目的操作数.

利用本文工具与 TEMU 在 12 个真实软件漏洞的对比检测结果表明:本文提出的污点分析方法更为完善,具备更强的漏洞检测能力和更高的分析效率,可检测和验证大量的已知漏洞.目前,本文工具已部署到检测 0day 漏洞的系统中.本文最后讨论了在实验过程中遇到的一些棘手的问题,这些问题可能导致污点信息的传播发生偏差,期望在以后的研究过程中逐步解决这些问题.

References:

[1] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697−1710 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01697]

[2] Luk C, Cohn R, Muth R, Harish P, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. of the ACM Conf. on Programming Language Design and Implementation. New York: ACM Press, 2005. 190−200. [doi: 10.1145/1065010.1065034]

[3] Lueck G, Patil H, Pereira C. PinADX: An interface for customizable debugging with dynamic instrumentation. In: Proc. of the IEEE/ACM Int'l Symp. on Code Generation and Optimization. New York: ACM Press, 2012. 114−123. [doi: 10.1145/2259016.2259032]

[4] Roy A, Hand S, Harris T. Hybrid binary rewriting for memory access instrumentation. In: Proc. of the 7th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: ACM Press, 2011. 227−238. [doi: 10.1145/1952682.1952711]

[5] Skaletsky A, Devor T, Chachmon N, Cohn R, Hazelwood K, Vladimirov V, Bach M. Dynamic program analysis of microsoft windows applications. In: Proc. of the Int'l Symp. on Performance Analysis of Software and Systems. Timisoara: IEEE, 2010. 389−400. [doi: 10.1109/ISPASS.2010.5452079]

[6] Patil H, Pereira C, Stallcup M, Lueck G, Cownie J. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In: Proc. of the 8th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. New York: ACM Press, 2010. 1020−1034. [doi: 10.1145/1772954.1772958]

[7] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang ZK, Newsome J, Poosankam P, Saxena P. BitBlaze: A new approach to computer security via binary analysis. In: Proc. of Int'l Conf. on Information Systems Security. Hyderabad: Springer-Verlag, 2008. 1−25. [doi: 10.1007/978-3-540-89862-7_1]

[8] Brumley D, Poosankam P, Song D, Zheng J. Automatic patch-based exploit generation is possible: Techniques and implications. In: Proc. of the IEEE Symp. on Security and Privacy. California: IEEE, 2008. 78−102. [doi: 10.1109/SP.2008.17]

[9] Yin H, Liang Z, Song D. HookFinder: Identifying and understanding malware hooking behaviors. In: Proc. of the 15th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2008. 103−119.

[10] Caballero J, Yin H, Liang Z, Song D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2007. 567−581. [doi: 10.1145/1315245.1315286]

[11] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proc. of the ACM Conf. on Computer and Communication Security. New York: ACM Press, 2007. 103−119. [doi: 10.1145/1315245.1315261]

[12] Kang M, Poosankam P, Yin H. Renovo: A hidden code extractor for packed executables. In: Proc. of the 5th ACM Workshop on Recurring Malcode. New York: ACM Press, 2007. 327−341. [doi: 10.1145/1314389.1314399]

[13] Ma JX, Li ZJ, Hu CJ, Zhang JX, Guo T. Research of array type abstraction reconstruction in binary code. Journal of Tsinghua University: Science and Technology, 2012,10(1):1329−1334 (in Chinese with English abstract). [doi: 10.16511/j.cnki.qhdxxb.2012.10.003]

[14] Ma JX, Li ZJ, Hu CJ, Zhang JX, Guo T. A reconstruction method of type abstraction in binary code. Journal of Computer Research and Development, 2013,50(11):2418−2428 (in Chinese with English abstract).

[15] Schwartz J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In: Proc. of the IEEE Security and Privacy Symp. 2010. 723−734. [doi: 10.1109/SP.2010.26]

[16] Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of the USENIX Symp. on Operating Systems Design and Implementation. Vancouver: USENIX, 2010. 817−822. [doi: 10.1145/2619091]

[17] Kang M, McCamant S, Poosankam P, Song D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2011. 913−926.

[18] Wang TL, Wei T, Gu GF, Zou W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability. In: Proc. of the 2010 IEEE Symp. on Security and Privacy. California: IEEE, 2010. 497−512. [doi: 10.1109/SP.2010.37]

[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 16th Network and Distributed System Security Symp. San Diego: Internet Society, 2010. 333−347.

[20] Hu CJ, Li ZJ, Guo T, Shi ZW. Detecting the vulnerability pattern of writing tainted value to tainted address. Journal of Computer Research and Development, 2011,48(8):1455−1463 (in Chinese with English abstract).

[21] Babak Y, Saumya D. Symbolic execution of obfuscated code. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2015. 732−744. [doi: 10.1145/2810103.2813663]

[22] Michelle W, David L. IntelliDroid: A targeted input generator for the dynamic analysis of android malware. In: Proc. of the 22nd Network and Distributed System Security Symp. San Diego: Internet Society, 2016. 481−496.

附中文参考文献:

[1] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1710. [doi: 10.3724/SP.J.1016.2009.01697]

[13] 马金鑫,忽朝俭,李舟军,张俊贤,郭涛.二进制代码中数组类型抽象的重构方法.清华大学学报:自然科学版,2012,10(1):1329−1334. [doi: 10.16511/j.cnki.qhdxxb.2012.10.003]

[14] 马金鑫,李舟军,忽朝俭,张俊贤,郭涛.一种重构二进制代码中类型抽象的方法.计算机研究与发展,2013,50(11):2418−2428.

[20] 忽朝俭,李舟军,郭涛,时志伟.写污点值到污点地址漏洞模式检测.计算机研究与发展,2011,48(8):1455−1463.

马金鑫(1986－),男,山西吕梁人,博士,副研究员,主要研究领域为软件安全,程序分析.

李舟军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为网络与信息安全,数据挖掘与智能信息处理.

张涛(1977－),男,博士,研究员,主要研究领域为软件安全,漏洞分析.

沈东(1989－),男,博士生,CCF 学生会员,主要研究领域为移动安全,虚拟化安全.

章张锴(1990－),男,博士生,主要研究领域为信息安全.