# 可控内存写漏洞自动利用生成方法

黄桦烽[1,2]，苏璞睿[1,2]，杨轶[1,2]，贾相堃[1,2]
（1. 中国科学院软件研究所可信计算与信息保障实验室，北京 100190；
2. 中国科学院大学计算机科学与技术学院，北京 100190）

**摘 要**：针对现有漏洞自动利用生成方法无法实现从“可控内存写”到“控制流劫持”的自动构造问题，提出一种可控内存写漏洞的自动利用生成方法。首先，基于内存地址控制力度的动态污点分析方法检测可控内存写漏洞；然后，基于漏洞利用模式进行利用要素搜索，通过约束求解自动构造可控内存写漏洞的利用。实验结果表明，所提方法可以有效检测可控内存写漏洞，搜索漏洞利用要素，自动生成从可控内存写到控制流劫持的利用。

**关键词**：可控内存写；控制流劫持；动态污点分析；漏洞利用要素；自动利用生成
**中图分类号**：TP311
**文献标志码**：A
**DOI:** 10.11959/j.issn.1000−436x.2022003

# Automatic exploitation generation method of write-what-where vulnerability

HUANG Huafeng[1,2], SU Purui[1,2], YANG Yi[1,2], JIA Xiangkun[1,2]
1. Trusted Computing and Information Assurance Laboratory, Institute of Software Chinese Academy of Sciences, Beijing 100190, China
2. School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100190, China

**Abstract:** To solve the problem that the current vulnerability automatic exploitation generation methods cannot automatically generate control-flow-hijacking exploitation from write-what-where, a method of automatic exploitation generation for write-what-where was proposed. First, the write-what-where vulnerability was detected based on the memory address control strength dynamic taint analysis method. Then, the vulnerability exploitation elements were searched based on the vulnerability exploitation modes, and the exploitation of write-what-where vulnerability was generated automatically by constraint solving. The experimental results show that the proposed method can effectively detect write-what-where vulnerability, search exploitation elements, and automatically generate the control-flow-hijacking exploitation from write-what-where.

**Keywords:** write-what-where, control flow hijacking, dynamic taint analysis, vulnerability exploitation element, automatic exploitation generation

## 0 引言

随着人工智能技术在安全漏洞领域的应用[1]，漏洞研究的自动化技术迅速发展。漏洞自动利用是探寻漏洞可能的利用方法，进而评估漏洞危害、防御漏洞利用攻击的重要手段。自 2016 年美国 DARPA CGC[2]（cyber grand challenge）比赛以来，漏洞自动利用已成为网络安全领域的研究热点。早期的 APEG[3]、AEG[4-5]、Mayhem[6]、PolyAEG[7]等在该领域进行了众多尝试，主要针对明确的控制流劫持类漏洞，并

**收稿日期**：2021−09−17；**修回日期**：2021−12−17
**通信作者**：苏璞睿，purui@iscas.ac.cn
**基金项目**：国家自然科学基金资助项目（No.U1736209, No.61572483, No.U1836117, No.U1836113, No.62102406）；中国科学院战略性先导科技专项基金资助项目（No.XDC02020300）
**Foundation Items:** The National Natural Science Foundation of China (No.U1736209, No.61572483, No.U1836117, No.U1836113, No.62102406), The Strategic Priority Research Program of the Chinese Academy of Sciences (No.XDC02020300)

仅在简单的程序上进行了验证，缺乏针对大规模软件、复杂漏洞类型的分析能力。

内存破坏漏洞是常见的漏洞类型。除了根据漏洞类型分类之外，从漏洞破坏能力刻画漏洞更有利于漏洞利用。内存破坏漏洞除了可能造成直接的控制流劫持外，也会表现为内存访问异常、段异常等，漏洞的终极破坏能力表现为任意代码执行。其中，可控内存写漏洞尽管起初可能只是造成程序崩溃，但具有很大的潜力转化为控制流劫持，最终造成任意代码执行。造成可控内存写漏洞的原因存在多种可能，包括数组索引边界检查缺陷、缓冲区溢出覆盖数据指针、UAF（use after free）造成指针误用、格式化字符串漏洞等情况；可控内存写漏洞的利用方式具有多样性，包括覆盖函数指针构造控制流劫持、覆盖权限变量实现提权、覆盖逻辑变量触发逻辑错误、覆盖读内存指针实现信息泄露等。实际上，模糊测试产生的大量 PoC（proof of concept）样本并不能直接造成控制流劫持或任意代码执行。在模糊测试发现的可控写错误 PoC 的基础上，分析并构造控制流劫持路径，对提高漏洞的自动利用能力，提升模糊测试漏洞挖掘结果的评估能力具有重要价值。

由于内存属性、复杂输入依赖关系及环境约束限制等因素，从可控内存写漏洞到控制流劫持、再到任意代码执行的漏洞利用仍是一个困难问题，当前仍没有完整的自动化方法。为此，本文提出了一种可控内存写漏洞自动利用生成方法，从可控内存写漏洞的判定、利用要素的自动化搜索、攻击链的自动构造方面展开研究，基于可控内存写漏洞的利用模式，设计并实现了可控内存写漏洞的利用自动生成原型系统，验证了本文方法的有效性，主要贡献和创新点如下。

1) 从漏洞破坏能力的维度建立了内存破坏漏洞利用能力的层次等级分类，包括崩溃异常、可控内存写、控制流劫持、任意代码执行 4 个层次的能力等级。

2) 提出了面向内存地址控制力度的动态污点分析方法，通过对指令控制力度的规则和语义分析，设计了面向内存地址控制力度的污点传播规则，提升了可控内存写漏洞判定的准确度，形成了可控内存写漏洞判定能力。

3) 分析总结了可控内存写漏洞的利用模式及利用要素，从代码、数据、函数指针等多个维度，分析了可控内存写漏洞利用的不同方式，实现了函数指针要素的自动搜索。

4) 实现了可控内存写漏洞自动生成控制流劫持利用的原型系统，针对测试集构造利用样本达到控制流劫持能力的数量比模糊测试提升 54%，显著提升了模糊测试结果的可利用性评估能力。

## 1 知识背景与相关工作

漏洞利用攻击是指攻击者利用程序缺陷构造输入，使程序在执行过程中执行了满足攻击者意图的代码。针对内存破坏漏洞，从漏洞破坏能力等级角度可将其分为崩溃异常、可控内存写、控制流劫持、任意代码执行 4 个层次。崩溃异常是指非预期地覆盖了其他变量导致内存非法访问异常；可控内存写是指程序写内存的地址和内容可被输入所控制，如指令 mov [edi], eax 中的 edi 和 eax 都受到输入控制；控制流劫持是指程序跳转的位置可以被输入修改和控制，如 call 指令的参数被输入所控制；任意代码执行是指程序执行的指令可以被修改为攻击者注入的攻击代码。这 4 种类型的能力从破坏程序运行到获得远程控制权逐层递进，通常能够获得的漏洞 PoC 属于崩溃异常，而漏洞利用以达到任意代码执行为目标，来获得最大攻击收益。

根据内存破坏漏洞不同的初始利用能力，内存破坏漏洞利用步骤可以进行分解和复用，如图 1 所示。左侧表示漏洞利用能力，中间是漏洞利用方法和步骤，右侧是漏洞利用缓解机制，箭头上的字母序号标注了该流程可能遭遇对应的漏洞利用缓解机制的对抗拦截。

漏洞利用生成需要对触发程序崩溃的 PoC 进行分析，动态监控程序解析 PoC 的过程，分析内存破坏漏洞初始的漏洞能力，结果如图 1 所示。进一步，根据不同漏洞能力的特点，选择不同的利用方法实施利用。内存破坏漏洞利用步骤总结如下。

1) 具备初始内存破坏能力条件下，尝试构造破坏内存中的数据指针变量，破坏的数据指针变量作为漏洞利用生成后续步骤的前提条件。

2) 修改被破坏内存中指针包括 2 种情况，第一种情况是该指针是函数指针，或者函数的返回地址，通过修改指针的内容，可以实现控制流劫持。针对此类指针的修改，操作系统的 RELRO（relocation read-only）等写保护机制会对该过程形成对抗。第二种情况是该指针是数据指针，通过修改指针内容，可尝试构造可控地址内存写。操作系

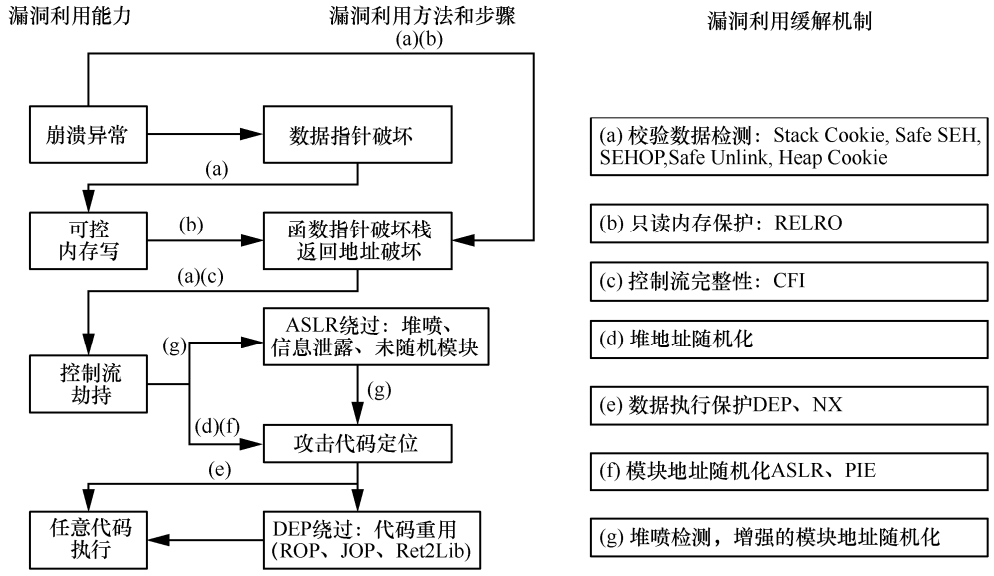


图 1　内存破坏漏洞利用步骤

统部署了 Meta Data、Canary、Safe Unlink、Heap Cookie 等防御机制，导致利用逻辑复杂、难度更高，本文研究针对数据指针的情况展开。

3) 在能够构造可控地址内存写的条件下，尝试定位并修改函数地址指针或者返回地址，如果该函数指针或返回地址在后续执行中被使用，则可构造控制流劫持。此部分需要从内存属性筛选出未受写保护的函数指针，避开 RELRO 写保护安全机制的影响。

4) 函数指针被覆盖到被引用触发控制流劫持过程中，中间可能受到 CFI（control-flow integrity）、Stack Cookie 等安全机制的影响，需要搜索控制流完整性和数据流完整性检验的缺陷和绕过空间。

5) 为了形成可用的控制流劫持，需要在内存中放置并定位 ShellCode，以确定控制流劫持方向。该过程可能受到 PIE（position independent executable）和 ASLR（address space layout randomization）等内存地址随机化影响，通过寄存器指针、栈空间数据指针搜索可尝试找出指向污点数据的指针，作为定位 ShellCode 位置的变量元素，此外，还可通过堆喷、信息泄露等方式提高 ShellCode 代码定位的成功率。

6) ShellCode 定位完成之后，结合控制流劫持可实现最终目标，即任意代码执行。但该过程可能受到 DEP（data execution prevention）和 NX（no execute）数据执行保护的影响，目前存在 ROP（return-oriented programming）、JOP（jump-oriented programming）、Ret2Lib（return-to-library）等代码重用的方式绕过数据执行保护，而代码重用过程中需要先定位出模块代码位置，这一过程采用的策略包括使用未随机化的代码模块、信息泄露模块地址，从这些模块搜索 ROP Gadgets 的代码片段，串接 ROP 功能实现利用，或者通过代码功能修改内存属性为可执行实现任意代码执行的效果。

以上阐述了内存破坏漏洞根据不同利用能力的漏洞利用生成方法和步骤，同时也阐述了相应过程可能遇到的安全机制限制，漏洞自动利用尝试将这些生成方法和步骤自动化实现。当前的研究工作仅实现了其中一部分环节的自动化，APEG[3]是基于补丁自动生成 Crash 的 PoC；AEG[4-5]、Mayhem[6]、PolyAEG[7]、CRAX[8]针对的是控制流劫持条件下的构造任意代码执行的漏洞自动利用；HCSIFTER[9]、Revery[10]、FUZE[11]针对的是堆类型漏洞，且只针对漏洞利用中的特定环节实现自动化；R2dlAEG[12]、FlowStitch[13]、DOP[14]、BOP[15]、Q[16]则是针对漏洞利用过程中的特定安全机制绕过环节实现了自动化。相关工作对应的漏洞自动利用生成环节如表 1 所示，序号含义为：①构造崩溃异常；②基于崩溃构造可控内存写；③基于崩溃构造控制流劫持；④基于可控内存写构造控制流劫持；⑤基于控制流劫持构造任意代码执行；⑥DEP 安全机制绕过；⑦CFI 安全机制绕过；⑧可利用性评估。

**表 1　相关工作对应的漏洞自动利用生成环节**

| 研究工作 | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
|---|---|---|---|---|---|---|---|---|
| APEG | √ | | | | | | | |
| AEG | | | | | √ | | | |
| Mayhem | | | | | √ | | | |
| PolyAEG | | | | | √ | | | |
| CRAX | | | | | √ | | | |
| HCSIFTER | | | | | | | | √ |
| Revery | | √ | √ | | | | | |
| FUZE | | | | | √ | | | |
| R2dlAEG | | | | | | √ | | |
| FlowStitch | | | | | | √ | √ | |
| DOP | | | | | | √ | √ | |
| BOP | | | | | | √ | √ | |

数据显示，现有研究还缺少可控内存写到控制流劫持利用阶段的自动化生成，同时该过程主要存在三方面的难点与挑战。

1) 在可控内存写漏洞的判定方面，依赖于动态污点分析对逐条指令的内存写操作进行分析，检测指令寻址寄存器是否受到输入控制，并且需要考虑控制的力度能否达到任意地址写的程度，写范围的限度是判断漏洞是否真实存在的难点。

2) 在可控内存写漏洞利用要素搜索方面，需要同时考虑函数指针或返回地址的定位，内存属性是否可写，以及函数指针是否能被引用的情况，即需要搜索出同时满足地址可定位、内存可写、会被调用这 3 个条件的函数指针。

3) 在控制流劫持触发方面，CFI、SHE（safe exception handler table）等安全机制的引入，可能导致被修改后的函数指针或者返回地址无法被调用执行，对虚函数表、SEH 表的完整性检查，以及 Stack Cookie 对栈溢出的检测都有可能拦截被破坏的函数指针及返回地址被引用，从而阻止控制流劫持，这些因素给可控内存写漏洞自动利用生成控制流劫持带来了诸多的挑战。

## 2　内存地址控制力度的动态污点分析方法

针对可控内存写漏洞自动判定的难点问题，动态污点分析是检测可控内存写漏洞的基本思路。其原理是标记用户可控的输入为污点，动态跟踪污点数据在程序运行过程中的传播过程，检测内存写入的指令参数是否受到输入控制进行该类型漏洞的判定。设计使用中涉及污点传播规则、检测规则和约束条件三方面。

在污点传播规则方面，基于传统污点分析的初步检测，本文发现其存在大量误报，通过对传播过程进行回溯分析得到传播路径，发现误报主要包括以下情况。

1) 污点数据作为堆分配大小，影响了下一个堆分配的位置，下一个堆的数据访问指针因受到污点数据影响导致误报。

2) 污点数据影响到内存访问地址，但只是部分的地址位，多次移位运算之后指针的多个比特位不受污点控制，并且污染的位置只是影响，达不到目标内存值任意控制的能力。

3) 数组访问的偏移索引虽然受到污点数据影响，但是其有前置约束条件，虽然实际情况是被污染，但达不到任意地址写破坏的效果。

总体而言是由于传统污点传播规则未区分控制计算和影响计算的程度，而地址计算依赖于严格的控制计算。虽然符号执行理论上可用于地址写控制范围的求解，但消耗大量的计算资源。针对这些问题，本文提出了面向内存地址控制力度的动态污点分析方法，通过对不同力度的计算指令污点传播规则进行改进，以适用于可控内存写漏洞的检测。

计算指令通常分为算术运算、逻辑运算、位运算等类型，地址计算涉及更多的是加减法运算，数组地址偏移量需要乘法运算支持多字节一组的单元寻址，但有时通过左移实现幂次倍的乘法运算，另外地址对齐也会进行移位或者与运算将低比特位抹零处理。针对这些特性，本文对相关指令的污点传播规则进行了策略修改，传播规则修改策略如表 2 所示。

**表 2　　传播规则修改策略**

| 指令 | 规则说明 |
|---|---|
| add | 单字节独立计算污点状态，不考虑低字节对高字节的进位污染，考虑了进位只能影响 1 bit，不能控制大范围地址 |
| sub | 单字节独立计算污点状态，不考虑低字节对高字节的借位污染，考虑了借位只能影响 1 bit，不能控制大范围地址 |
| lea | 单字节独立计算污点状态，不考虑低字节对高字节的进位污染，考虑了进位只能影响 1 bit，不能控制大范围地址 |
| and | 与运算的非污点字节如果比特数超过 6 bit 为 0 对该字节漂白，通常用于内存页对齐，失去了可控计算的能力 |
| or | 或运算的非污点字节如果比特数超过 6 bit 为 1 对该字节漂白 |
| shl | 左移位数超过 6 bit 的低字节漂白，通常用于内存页地址对其，失去了可控计算的能力 |

在检测规则方面，可控地址写的检测需要指针 4 byte 受到控制才具备任意地址写的能力，考虑到偏移地址受控情况可能低 2 byte 被控制即可实现破坏，本文检测低 2 byte 受污染对可控内存写进行判定。比如 mov [edx], eax 指令，需要 edx 的低 2 byte 受到污点控制，发现误报的案例中地址指针的低字节并不受污点控制，其中大量使用了 shl 左移指令计算内存页地址，地址访问只是影响了所在内存页，低地址不受污染，并不具备控制地址写能力。

在约束条件方面，本文针对 cmp 比较指令进行了记录，如果指针引用前做了边界检测将进行记录，根据上下文可以判定出警告的可控地址写指针是否有安全边界检测，如果边界检测的点与可控地址写的告警点位于同一个函数上下文，且相关变量污点源相同，可初步判断告警是误报情况。

## 3　系统设计与实现

针对当前可控内存写漏洞构造控制流劫持利用的难点与挑战，本文设计实现了可控内存写漏洞自动利用生成方法，其流程如图 2 所示。

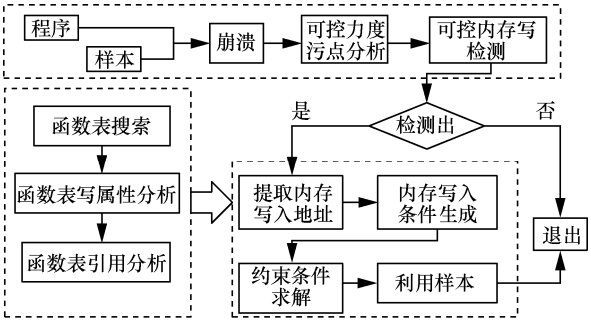


图 2　可控内存写漏洞自动利用生成方法流程

首先，基于可控力度污点分析，判定其是否达到可控内存写漏洞的利用能力；然后，基于静态分析和动态分析，提取出程序中的函数表，包括导入表、虚函数表、回调函数表等，筛选出其中具备写权限的部分，并以动态运行的方式筛选出可控内存写触发点之后潜在调用执行的点；最后，提取备选覆盖的内存写入地址，及覆盖内存的约束条件和内容，由输入求解引擎进行约束条件求解，生成漏洞利用样本。

本文实现了可控内存写漏洞的自动利用原型系统，原型系统框架如图 3 所示，包含基于 QEMU 实现动态插桩提取指令记录和污点源，基于 Udis86 反汇编和可控内存力度污点传播规则实现可控内存写漏洞的自动检测，基于指令记录分析和静态分析提取代码、数据、函数指针、数据指针、内存属性的利用要素，基于利用模式提取利用约束并采用 Z3 求解引擎对输入进行求解。

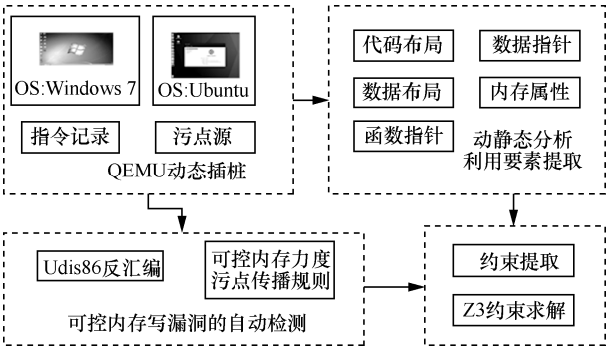


图 3　原型系统框架

本节从可控内存写漏洞的判定、可控内存写漏洞利用模式及要素自动搜索、可控内存写漏洞利用自动生成三方面对本文系统设计与实现进行详细介绍。

### 3.1　可控内存写漏洞的判定

可控内存写漏洞的判定使用了文献[17]提出的 Weak-Tainted 程序动态运行时漏洞检测模型，基于该模型对脆弱点和污点属性进行定义，可控内存写漏洞检测模型如图 4 所示，脆弱点集合包含 2 种类型：Weak-INS 和 Weak-PC。Weak-INS 是指可能引起可控内存写能力的指令模式，例如 mov [edi]、eax、rep movsd 等；Weak-PC 是指潜在可控内存写能力的函数地址集合，通过静态分析提取敏感脆弱函数地址，例如提取 memcpy、strcpy、scanf、sprintf、read 等具备写内存能力的函数，通过静态解析 libc 库的导出函数表获得函数偏移地址，结合模块基地址计算得到运行过程的函数地址。污点属性需要给脆弱点集合的每个脆弱点定义污点检测规则，检测相应寄存器等变量是否受输入污染。不同脆弱点都有对应的存储单元检测规则，当脆弱点定义的检测变量受到输入污染，则可筛选出具备可控地址写能力的脆弱点。

与控制流劫持检测相比，可控内存写的检测资源消耗会更高。从指令的层面分析，控制流劫持检测只需要检测转移跳转指令，而可控内存写的检测需要检测所有内存写操作的指令，这类指令所占比例更高，所需消耗的计算资源更多。但在动态污点分析过程中，本身就需要对指令进行反汇编，提取操作数类型，计算操作数地址，这些过程的结果可以直接复用在可控内存写检测当中，从而减轻额外的计算资源开销。

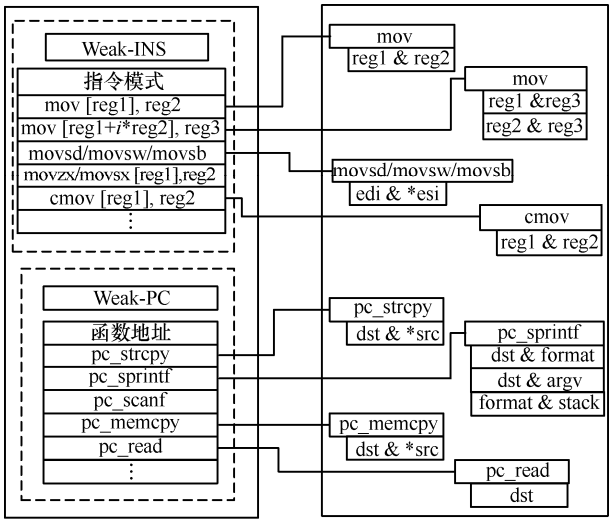


图 4　可控内存写漏洞检测模型

本文基于内存地址控制力度的动态污点分析方法，将文件、网络、键盘等输入标记为污点源，并赋予每个输入字节不同的标签，动态跟踪解析执行的每条指令，分析指令对“污点数据”的“传播”和“清除”过程，动态更新寄存器和内存的污点状态及其对应的污点标签。

可控内存写漏洞检测流程如图 5 所示。1) 将二进制程序和 PoC 输入置入 QEMU 中动态加载运行，通过 QEMU 对进程实现动态插桩，根据指令及 API 参数判定是否引入新的污点源，如果引入新的污点源，则更新记录污点状态的结构 Taint Map；2) 对指令进行反汇编解析，根据指令语义解析进行动态污点传播分析，检索源操作数的污点状态，更新目标操作数的污点状态及计算新的污点标签；3) 在动态污点分析计算指令操作地址过程中，分析操作数类型，当目的操作数的类型为内存时，进入可控内存写检测流程，否则进入下一步；4) 当指令写操作数的地址为内存时，如果是立即数寻址则可直接跳过，如果是寄存器寻址则检测直接寻址和间接寻址的寄存器是否为污点，如果为污点再检测数据源是否为污点，两者同时为污点的情况下则具备了可控内存写可控内容的能力；5) 针对具备可控内存写能力的潜在点，记录输出报告信息，内容包括指令地址、指令内容、寻址寄存器（地址指针）、寻址寄存器的污点源、源数据、源数据对应污点源；6) 循环分析下一条指令。

### 3.2　可控内存写漏洞利用模式及要素自动搜索

可控内存写漏洞具有较强的破坏能力，单指令层面的可控内存写具备篡改单字节、双字节、四字节或八字节的能力，例如 mov [edi]、al、ax、eax、

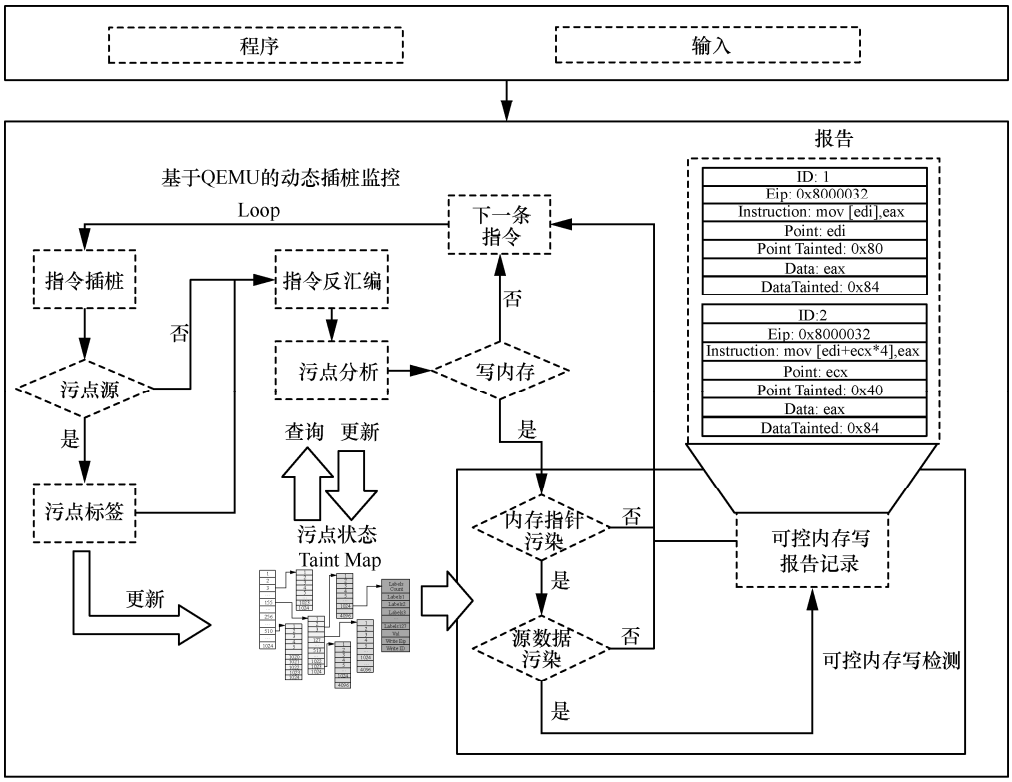


图 5　可控内存写漏洞检测流程

rax；函数层面的可控内存写具备篡改更多长度字节的能力，例如 read（dst, fd, size）、memcpy（dst, src, size）等。攻击者在获得可控内存写漏洞能力的前提下，可以肆意破坏内存，篡改包括指令、数据、指针、函数等类型的变量。与控制流劫持漏洞相比，可控内存写漏洞利用模式更加丰富，如图 6 所示。

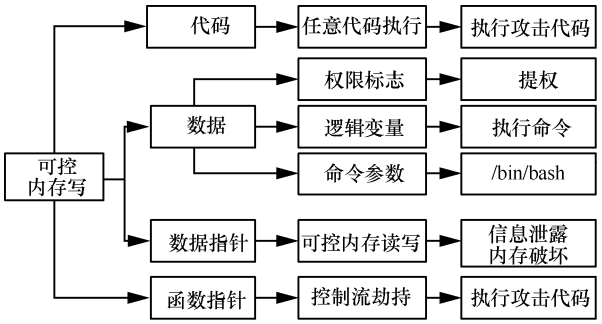


图 6　可控内存写漏洞利用模式

篡改指令可以直接诱发任意代码执行，但正常情况下指令所在内存处于不可写状态，不排除 SMC 等动态代码的情况；篡改数据需要根据数据类型而定，例如篡改权限标记变量可达到提升权限能力，篡改逻辑变量可触发逻辑错误执行本不具备权限的代码，篡改 system 等敏感函数参数可触发命令注入执行；篡改数据指针可诱发二次的可控内存读写，构造更多的数据破坏；篡改函数指针可触发控制流劫持，构造任意代码执行的漏洞利用。

基于可控内存写漏洞利用模式，本文设计实现了其中通过劫持函数表和函数指针实现控制流劫持的利用生成方式，具体包括内存读写属性的提取、函数表地址的动态提取与静态提取、潜在利用可能函数表的筛选。

内存的读写执行权限属性是极关键的前提要素，可控内存写需要写入的内存地址具备写权限。由于现代操作系统增强了内存的安全管理，在进程初始化完成之后对进程空间的代码区域、动态库导入表等敏感内存位置，设置了只读属性，这些内存区域在可控内存写利用生成过程中无法作为被写入的内存区域。因此在漏洞利用生成过程中，需要通过内存读写属性的分析，筛选其中具备写权限的内存，再筛选满足其他条件的利用要素。

内存权限属性的提取可以通过内存页属性查询进行解析，Windows 系统提供了 VirtualQueryEx 内存属性查询接口，输入参数包括句柄 HANDLE、查询的内存地址、接收返回值的结构地址和大小，输出为 MEMORY_BASIC_INFORMATION 结构体信息，其中 BaseAddress 为查询地址所在内存块的基地址，RegionSize 为内存区块大小，Protect 为权限属性，通过循环解析可以获取整个进程空间的内存属性，进程空间内存可写权限属性提取伪代码如算法 1 所示。

**算法 1**　进程空间内存可写权限属性提取伪代码

**输入**　进程句柄 hPro

**输出**　可写内存属性集合 MSet{$Set_1$,$Set_2$,…, $Set_n$}

```
1) Addr = 0
2) MEMORY_BASIC_INFORMATION mbi;
3) while Addr
4)   VirtualQueryEx(hPro,addr,&mbi, sizeof(mbi))
5)     if (mbi.Protect & WRITE_MASK):
6)       Set = (mbi.BaseAddress,mbi.ReginSize,
         mbi.Protect)
7)       MSet.insert(Set)
8)     end if
9)     Addr = mbi.BaseAddress + mbi. ReginSize
10) end while
```

函数指针及函数表的覆盖是构造控制流劫持的关键步骤，传统漏洞是基于缓冲区溢出覆盖函数表或者函数返回地址，溢出覆盖函数表依赖于缓冲区与函数表位于同一结构体或者临近的结构体。可控内存写漏洞则可以选择性地覆盖备选的函数指针或者函数返回地址，但由于函数返回地址所处栈空间基本随机，难以直接定位。函数表或者函数指针的地址可定位能力相对较强，除了函数导入导出表可供选择之外，虚函数表、回调函数指针、跳转表也都是备选的覆盖项。

备选覆盖函数指针的筛选依据包括函数表所在内存可写、函数指针在触发可控写内存漏洞之后会被调用、函数表地址可定位。为了统计出满足条件的函数或者跳转表，本文基于动态指令分析收集备选的函数表利用要素。

首先基于动态插桩获取执行的指令记录，反汇编筛选其中的函数调用指令 call 和跳转指令 jmp 进行分析。call/jmp 指令的跳转方式包括间接跳转和直接跳转，直接跳转是通过指令地址的相对偏移实施跳转，由于不存在可变参数，不会出现劫持情况。间接跳转是通过寻址方式计算获得跳转地址，其潜在跳转地址被劫持可能，同时也包括寄存器跳转

call/jmp eax 和内存地址跳转 call/jmp [eax+0x20]2 种情况。调用寄存器的情况需要分析目标寄存器源自的内存地址，调用寄存器代码示例如图 7 所示，0x000187A6 处的 call ecx 往上分析追溯到 0x00018794 处的 mov ecx, [eax+0xc]，如果可控内存写漏洞对内存[eax+0xc]进行了覆盖，会导致 call ecx 位置出现控制流劫持。

```
·text:00018794    loc_18794:                                  ; CODE
·text:00018794                    mov     ecx, [eax+0ch]
·text:00018797                    test    ecx, ecx
·text:00018799                    jz      short loc_187B0
·text:0001879B                    mov     [esp+6Ch+var_50], eax
·text:0001879F                    mov     [esp+6Ch+env], edx
·text:000187A2                    mov     [esp+6Ch+var_54], edx
·text:000187A6                    call    ecx
·text:000187A8                    mov     eax, [esp+6Ch+var_50]
·text:000187AC                    mov     edx, [esp+6Ch+var_54]
```

图 7　调用寄存器代码示例

因此，针对调用寄存器的情况，需要逆向分析出寄存器来源的内存地址作为可控内存写的利用备选地址，而类似 call [0x6583138c]、call[eax+0x20]这类的指令可直接获取内存地址作为备选地址。

为了解决寄存器作为调用参数的内存源地址追溯问题，传统思路是针对寄存器做逆向切片分析，本文采用了短距离的动态污点分析，在基于污点分析检测可控内存地址写的过程当中，针对内存数据到寄存器的拷贝过程，记录了寄存器来源的内存地址，并作为污点标签在短距离内进行计算传播，短距离污点传播伪代码如算法 2 所示。之所以称为短距离是因为寄存器的更新频率非常高，当采用新的内存数据对寄存器进行覆盖时，立即更新寄存器的污点标签为新的内存地址标签，同时忽略寄存器到内存、内存到内存直接的传播，只是为了满足寄存器溯源内存而提出的微缩版的动态污点分析方案。

**算法 2**　短距离污点传播伪代码

**输入**　指令序列

**输出**　call 指令寄存器的污点状态

```
1) for ins in instruction_lists
2)    (OP_DST, OP_SRC, OP_CODE) = Disas
      sembly(ins);
3)    if OP_DST is Reg and OP_SRC is Memory
4)       OP_DST.SetTaintLabel(Memory_Addr)
5)    else if OP_DST is Reg and OP_SRC is Reg
6)       OP_DST.SetTaintLabel(GetTaintLabel(OP_
         SRC))
7)    end if
8)    if OP_CODE is call and OP_DST is Reg
9)       MemSrcAddr = GetTaintLabel(OP_DST)
10)      Output MemSrcAddr
11)   end if
12) end for
```

另外，存在一类函数调用的函数指针无法通过指令记录分析获得，但是这类函数却具有被调用的可能。该类型普遍用于自定义 hook 库函数接口的结构指针变量，初始值为 0，如果被赋值为非 0 则会被调用，包括 lib_hook_malloc、lib_hook_free 等，遗漏函数模式代码示意如图 8 所示。指令记录中，由于函数指针变量未被覆盖，不会触发调用该函数指针的代码，上述代码中的 lib_hook_func 函数不会被调用，相应的汇编指令 call eax 也不会被执行。

```
if(lib_hook_func)
{
    lib_hook_func(argv);
}
```

```
mov eax, ds:dword_80ed650
test eax eax
jz short loc_805a908
call eax
```

图 8　遗漏函数模式代码示意

针对这种情况，本文采用静态模式匹配的方式，基于 libc 库中存在该模式的函数提取特征，在二进制程序中搜索匹配相应的函数，识别函数之后提取函数表的全局变量地址。

潜在调用点函数通过其在指令记录中的出现频度和出现顺序进行排序筛选，通过构造一个不触发崩溃的样本，在检测出的可控内存写之后的点分析被引用的函数表或函数指针，以及选择调用频度高的函数指针作为备选的劫持点。

### 3.3　可控内存写漏洞利用自动生成

可控内存写漏洞利用的自动生成依据漏洞判定给出的漏洞点和输入依赖字节，结合漏洞利用模式及利用要素的自动提取筛选出备选的内存写入点，构造出覆盖函数指针的约束求解条件。约束条件主要包括 2 个：1) 漏洞点写入的内存地址约束其等于备选利用方式需要写入的内存地址；2) 漏洞点写入的内容约束其等于备选利用方式预期的内存值。

针对指令检测出的可控内存写，其利用生成的约束条件是清晰且明确的。针对函数层面的检测出的可控内存写，其利用生成的约束条件依据函数的复杂程度而定，类似 memcpy、strcpy、strncpy 这类写入内存地址和内容参数变量明确的情况，其约束

条件也是清晰明确的。但是，类似 sprintf、fprintf、printf 这类格式化字符串函数，当格式化参数 format 受到控制时，通过控制栈上的指针和格式化参数同样可以构造可控内存写，但是其约束条件的构造则比较复杂。

格式化字符串漏洞在可控内存写利用能力构造的约束条件生成利用了格式字符%n 的功能，该功能是向栈指针指向的内存地址写入前序输出的字符数量。以 32 bit 程序为例，%n 是写入 4 byte 内存，%hn 是写入 2 byte 内存，%hhn 是写入 1 byte 内存。此外，格式化字符串的单次输出字符串数量上限是 65 535，因此如果要写入 4 byte 的地址空间，需要拆成 2 次的 2 byte 写入，即使用%hn 的格式字符进行写入。同时，写入的内存地址是通过栈上的变量指针进行寻址，因此需要在栈上搜索出 2 个 4 byte 被污点输入控制的变量，约束这 2 个变量为写入目标内存的地址 addr_for_write 和 addr_for_write+2。如图 9 所示，格式化字符串函数 printf 的 format 参数被输入控制，利用目标是向 addr_for_write 写入 4 byte 的 value，利用过程如下。

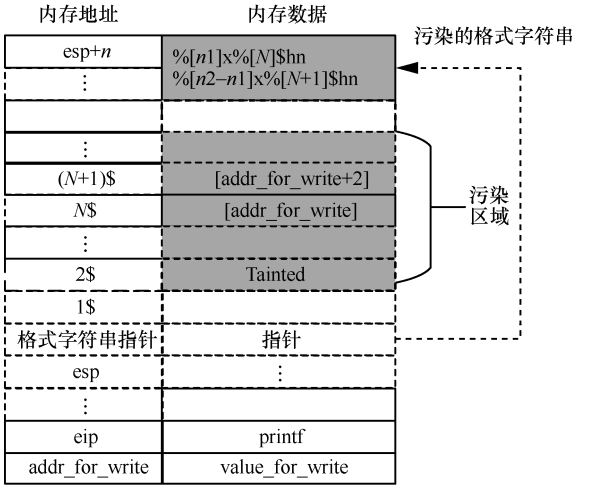


图 9　格式化字符串的可控内存写生成

1) 先将 value 分解成低高 2 个 word 字段 $n$=value&0xffff 和 $m$=(value>>16)&0xffff, 需分别往 addr_for_write 和 addr_for_write+2 写入 $n$ 和 $m$，由于输出字符数量是递增，因此 2 次写入的字节只能先写入小值，再写入大值。

2) 在栈上搜索出 2 个 4 byte 被污点输入控制的变量，计算与 format 指针参数的位置距离，距离单位是参数个数，因此 $N$= [taint_addr-point_addr]/4，图 9 中 2 个变量是连续的，因此分别为 $N$ 和 $N$+1。

3) 构造格式化参数，$m$>$n$ 则构造格式化字符串“%[$n$]x%[$N$]$hn%[$m-n$]x%[$N$+1]$hn”（方括号内的值需要写入实际计算得到的 10 进制数值），$m$<$n$ 则构造“%[$m$]x%[$N$+1]$hn%[$n-m$]x%[$N$]$hn”。

4) 根据实际情况整合约束条件，如 $m$>$n$ 的情况下得到约束条件，最后根据约束求解方法对约束条件函数进行求解，函数公式中 format 简写为 fmt。

$$\begin{cases}[N\$] == \text{addr\_for\_write} \\ [(N+1)\$] == \text{addr\_for\_write+2} \\ [\text{fmt}] == \%[n]\text{x}\%[N]\$\text{hn}\%[m-n]\text{x}\%[N+1]\$\text{hn} \end{cases}$$

对约束条件的求解，除了约束条件还需要有约束表达式，本文通过动态污点分析已经能够提取漏洞点约束条件中的输入相关变量，在此基础上，将约束条件相关的变量符号化，重新解析动态插桩过程提取的指令记录，根据初始的符号化变量，采用 Z3 的语法规则生成计算表达式，并自动提取 python 格式的表达式计算过程，自动生成对约束条件的求解脚本，最后执行 Z3 求解脚本生成利用的 exp 样本。

## 4　实验验证与结果分析

本文选取了 2018 年 Defcon China CTF 比赛中的 50 道 Linux 题目和 Windows 系统下的 PLC WinNT、Media Player Classic、DirectShow Player GUI、WINWORD、VUPlayer、Float Ftp 作为测试集。选取的真实程序包括了工控程序、办公软件、多媒体软件和网络传输工具等，并根据 PoC 要求选择了软件版本，其中 PLC WinNT 是 SCADA 工业控制软件代表 CODESYS 中的重要组件。考虑到不同版本软件在功能和复杂度上处于相同量级，且相关版本软件仍在广泛使用，本文选取的程序具有一定的代表性。

### 4.1　可控内存写检测分析

本文实现的可控内存写检测基于动态污点分析实现，针对 CTF 测试用例，将标准输入 stdin 标记为污点，采用 qemu 的用户态模式进行指令插桩，基于 UDIS86 反汇编引擎解析指令实现动态污点传播规则，针对写内存的指令检测内存写入地址的寻址寄存器是否为污点判定是否潜在可控内存写。

CTF 赛题测试集的 PoC 输入是通过 AFL++模糊测试引擎获得的，选择能够触发可控内存写或控制流劫持的 PoC 作为测试本模块的数据用例。测试结果如表 3 所示，50 个用例中有 22 个样本的利用能力达到控制流劫持，16 个达到可控内存写；另外

pwn12 的数据传播方式通过控制字段编码，污点分析传播会断层判定失败，pwn32 是命令注入类型漏洞，不属于内存破坏漏洞，另外几个用例 Fuzz 得到的 PoC 未能触发这 2 种状态。

**表 3 测试结果**

| pwn 样本集编号 | 判定情况 | 统计数/个 |
|---|---|---|
| 01,02,04,05,07,10,13,18,22,24,25, 26,27,30,31,34, 37,38,45,48,49,50 | 控制流劫持 | 22 |
| 09,14,15,17,19,20,21,23, 28,35,36,39,40,41,42,46, | 可控内存写 | 16 |
| 32 | 命令注入 | 1 |
| 03,06,08,11,12,16,29,33, 43,44,47 | 未能判定 | 11 |

此外还有 7 个样例是格式化字符串漏洞，无法通过指令层面进行判定，即使格式化字符串漏洞的 PoC 注入了%n 的可控地址写功能，也无法通过 mov [taint], taint 污点模式进行判定，这是由于其写入的值是前序输出的字符数量，该参数不具备 taint 属性，而只能是 mov [taint], value 模式。针对该问题，本文通过检测函数的格式化参数污点属性及栈指针污点属性，实现了基于函数层面的可控内存写漏洞利用能力判定。函数级格式字符串漏洞识别情况如表 4 所示，均能准确识别出格式化字符串漏洞。

**表 4 函数级格式字符串漏洞识别情况**

| 样本 | 格式化函数 | 能否检测 | 利用方式 |
|---|---|---|---|
| Pwn06 | sprintf | 能 | 溢出 ret |
| Pwn11 | snprintf | 能 | 可控内存写 |
| Pwn16 | printf、sprintf | 能 | 溢出 ret |
| Pwn29 | printf | 能 | 信息泄露覆盖 ret |
| Pwn33 | printf | 能 | 可控内存写 |
| Pwn43 | printf | 能 | 可控内存写 |
| Pwn47 | snprintf | 能 | 可控内存写 |

同时，函数层面的可控内存写判定还考虑 read 函数，该函数直接调用系统调用将数据读入指定的内存缓冲区，如果指定内存的指针受到输入控制同样会触发可控内存写。

另外，针对现实场景的应用程序，本文选取的 PLC WinNT、Media Player Classic、DirectShow Player GUI 是存在可控内存写漏洞的实例程序，给予其触发可控地址写的 PoC 作为测试输入，测试本文方法能否检测出崩溃点存在可控地址写的能力；WinWord、VUPlayer、Float Ftp 程序是测试控制流劫持漏洞的样本，评估在具备检测控制流劫持条件下是否有可控内存写的误报情况。

基于内存地址控制力度的动态污点分析方法，实验测试了 6 款软件，实例程序可控内存写检测情况如表 5 所示。从表 5 可知，能够有效检测出 6 款软件的漏洞 PoC 其中 3 款具备可控内存写，3 款具备控制流劫持的情况，同只出现了 PLC WinNT 的一处误报，该误报符合文中所述的指针索引虽被污染但受约束情况，能够被有效发现是误报，同时未出现漏报情况。

**表 5 实例程序可控内存写检测情况**

| 实例程序 | 版本 | “可控内存写”告警数量 | | |
|---|---|---|---|---|
| | | 告警数/个 | 误报数/个 | 漏报数/个 |
| PLC WinNT | V2.4.7.52 | 2 | 1 | 0 |
| Media Player Classic | 1.3.1249.0 | 1 | 0 | 0 |
| DirectShow Player GUI | V1.0 | 1 | 0 | 0 |
| WinWord | 14.0.4760.1000 | 0 | 0 | 0 |
| VUPlayer | 2.49 | 0 | 0 | 0 |
| Float Ftp | 1.0 | 0 | 0 | 0 |

本节实验验证了基于污点分析进行可控内存写漏洞检测的有效性，针对 CTF 类型的小程序只需采用传统的污点传播规则，针对真实的复杂程序传统的污点传播规则会出现误报，本文提出的内存地址控制力度动态污点分析方法能够有效消除误报，同时结合上下文约束条件能够有效发现误报情况。

### 4.2 漏洞利用要素自动提取分析

本文实现了可控内存写漏洞函数指针及函数表利用要素的字段搜索和筛选。基于动态内存页分析提取内存页读写执行权限属性，基于动态指令分析监测函数指针及跳转表的引用情况，统计出其中潜在引用可能的函数表属性情况，包括函数表不可写数量、函数表可写数量、函数表地址未随机且可写的数量。

结合静态分析提取 malloc 和 free 函数中潜在的 hook 函数指针作为注入点，搜索统计了 50 个测试用例的函数表要素情况。CTF 测试集函数指针利用要素统计情况如图 10 所示，48 个用例都存在函数指针利用要素，仅 pwn34 和 pwn45 这 2 个测试用例未发现函数指针利用要素。

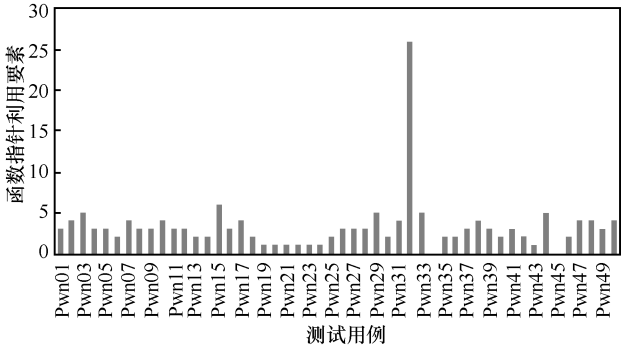


图 10　CTF 测试集函数指针利用要素统计情况

本文搜索统计了 6 款实例程序漏洞利用函数表要素提取情况，如表 6 所示。其中，UW 表示不可写函数表数量，AW 表示可写函数表数量，UAW 表示地址未随机且可写的函数表数量。通过统计可以发现，大型程序中函数表的使用很普遍，其中可改写的函数表也占用了一定比例，这是由于 C++程序使用虚函数指针，是不可避免的，地址随机化一定程度上缓解了函数指针被定位的问题，但是通过本文的搜索发现仍然有很多程序和模块未使用地址随机化对程序进行保护，因此本文仍能在测试用例中搜索出未随机化的可改写的函数表。

**表 6　实例程序漏洞利用函数表要素提取情况**

| 实例程序 | 版本 | 函数表利用要素统计 | | |
|---|---|---|---|---|
| | | UW/个 | AW/个 | UAW/个 |
| PLC WinNT | V2.4.7.52 | 1 067 | 81 | 54 |
| Media Player Classic | 1.3.1249.0 | 2 507 | 925 | 17 |
| DirectShow Player GUI | V1.0 | 707 | 954 | 26 |
| WinWord | 14.0.4760.1000 | 3 969 | 1 012 | 1 |
| VUPlayer | 2.49 | 162 | 11 | 10 |
| Float Ftp | 1.0 | 637 | 334 | 27 |

此外，通过实验统计的原始数据，本文还发现同一位置指令存在多次执行的情况，多次执行引用的内存表地址可能相同也可能不同，例如 call [ebp+0x8] 指令，多次执行到该位置时，由于 ebp 值的差异使引用的函数表位置不一致。针对引用相同位置的情况，统计进行了去重；针对引用不同位置的情况，统计进行了累加计算。另外，如果是不同位置引用了同一位置的函数表，统计也按照去重的方式进行处理，这是因为本文以收集函数表利用要素为导向，相同位置的函数表只能作为利用要素的一个原子。

### 4.3　可控内存写漏洞利用自动生成

针对 50 个测试用例中通过指令或函数判定为存在可控内存写能力漏洞的 20 个测试用例，本文基于函数利用要素的搜索获得需要写入的地址，通过可控内存写判定时的污点数据关系自动生成约束求解条件，并进行约束求解。可控内码写漏洞利用生成情况如表 7 所示。

**表 7　　可控内存写漏洞利用生成情况**

| 测试用例 | 利用点 | 构造情况 |
|---|---|---|
| Pwn09 | 0x407ffedc | 失败 |
| Pwn11 | 0x080ec4f0 | 失败 |
| Pwn14 | 0x080ec4f0 | call 劫持 |
| Pwn15 | 0x080f14f0 | 失败 |
| Pwn17 | 0x080ed4f0 | 失败 |
| Pwn19 | 0x080ec4f0 | call 劫持 |
| Pwn20 | 0x080ec4f0 | call 劫持 |
| Pwn21 | 0x080ec4f0 | call 劫持 |
| Pwn23 | 0x407ffdcc | 失败 |
| Pwn28 | 0x080ec4f0 | call 劫持 |
| Pwn33 | 0x080ecb60 | 失败 |
| Pwn35 | 0x080ec4f0 | call 劫持 |
| Pwn36 | 0x407ffedc | ret 劫持 |
| Pwn39 | 0x80f1b80 | call 劫持 |
| Pwn40 | 0x080ec810 | call 劫持 |
| Pwn41 | 0x080ec4f0 | call 劫持 |
| Pwn42 | 0x080eb4d8 | 失败 |
| Pwn43 | 0x080ed650 | 失败 |
| Pwn46 | 0x080ede60 | call 劫持 |
| Pwn47 | 0x080ec4f0 | call 劫持 |

利用点标记了生成过程中选取的写入内存地址，构造情况记录了利用生成达到的效果。表 7 中的利用生成存在 1 个 ret 劫持，这是通过控制地址的栈偏移指针，构造覆盖函数返回地址实现 ret 控制流劫持；11 个 call 劫持有 9 个是通过绝对地址进行定位覆盖 free 函数的 hook 点构造控制流劫持，剩下的 2 个(pwn39,pwn46)是覆盖了程序自定义的函数指针实现劫持。构造失败的用例情况包括：1) 可控内存写指令每次只覆盖 1 byte，需要多次构造可控内存写，自动化构造失败；2) 需要输入负整数经 atoi 变换后溢出，自动求解失败；3) 程序是多进程，分析框架支持能力受限。

另外，针对 PLC WinNT 实例程序中的 81 个可写函数表，本文搜索筛选出了触发可控内存写后潜在的 42 个引用点，其中 38 个是通过 call 内存模式，

包括 call [IMM]类型和 call [eax+IMM]类型，4 个是通过 jmp 内存模式，均为 jmp [IMM]模式（IMM 指立即数常量）；同时，本文检测出这些函数表和跳转表是具备内存写权限的，能够基于可控内存写的方式对其进行篡改，从而构造控制流劫持。通过利用生成，针对其中的 35 个利用点，成功触发了控制流劫持，另外 7 个未能触发控制流劫持是由于地址是随机的，无法在实际运行过程定位绝对地址。

实验表明，本文提出的可控内存写漏洞自动利用方法能够有效基于可控地址写构造控制流劫持漏洞利用能力，50 个测试用例中，基于 AFL 模糊测试挖掘的漏洞仅有 22 个达到控制流劫持状态，通过本文方法增加了 12 个达到控制流劫持状态，提升了 54%。同时，针对 PLC WinNT 的可控地址写漏洞，构造了 35 个不同位置的控制流劫持，验证了本文方法在实际漏洞程序中的有效性。

## 5 讨论

本文提出的可控内存写漏洞自动利用生成方法针对触发可控内存写漏洞样本，构造能够触发控制流劫持的 PoC，进一步提升验证了漏洞的破坏能力。而 Mayhem 与 PolyAEG 的相关工作仅支持控制流劫持类型的漏洞，在 PoC 未达到控制流劫持状态情况下是失效的。本文方法利用通过可控内存写漏洞的自动判定、漏洞利用要素的自动搜索和约束条件的自动提取与求解实现利用样本自动生成，补充了现有 AEG 方案的不足。

本文提出了内存地址控制力度的动态污点分析方法，与 libdft、Triton、Decaf 等污点分析方法相比，对污点传播规则进行了适配性的优化改进，消除了可控内存写漏洞判定的大量误报情况，而传统的污点传播规则带来的大量误报会导致指令级的检测无法正常使用。同时，本文方法也具备多标签污点标记能力，能够直接定位出输入关联的字节位置，服务于控制流劫持样本的自动构造。

另外，本文总结了可控内存写漏洞利用的多种模式及相关要素，针对构造控制流劫持的模式实现了漏洞利用函数表要素的自动化搜索，通过指令记录结合短距离污点分析的方式提取潜在可利用的函数表，同时基于内存读写属性筛选出其中可被改写的函数表，又结合模块地址随机化属性提取出其中的地址固定的部分，有效筛选出真正可被利用的函数表。针对指令记录无法获取的函数表，本文针对已知动态库中的代码特征模式进行特征搜索，提取了其中的一部分。但是，这类函数表的搜索提取并不完整，结合模糊测试挖掘更多的潜在路径、提取更完整的利用要素将是后续的研究点。

本文提出的可控内存写漏洞自动利用生成方法仅针对 Linux 和 Windows 用户态程序实现了原型系统，研究表明可控内存写漏洞在内核漏洞中频繁出现，且具有很强的破坏能力，本文方法在内核漏洞的应用和推广也将作为后续的研究点。

## 6 结束语

本文提出了一种可控内存写漏洞的自动利用生成方法，基于内存地址控制力度的动态污点分析方法从指令和函数层面识别判定可控内存写漏洞，结合动态插桩分析提取构造控制流劫持漏洞利用的要素，通过污点分析关联的漏洞点自动提取漏洞利用生成的约束条件，最后自动进行约束求解生成利用样本。通过对 50 个测试用例和 PLC 等实例程序进行测试评估，验证了本文方法的有效性，生成利用样本达到控制流劫持能力的数量比模糊测试提升了 54%，提升了漏洞的可利用性评估能力。

## 参考文献：

[1] 孙鸿宇, 何远, 王基策, 等. 人工智能技术在安全漏洞领域的应用[J]. 通信学报, 2018, 39(8): 1-17.
UN H Y, HE Y, WANG J C, et al. Application of artificial intelligence technology in the field of security vulnerability[J]. Journal on Communications, 2018, 39(8): 1-17.

[2] SONG J, ALVES-FOSS J. The DARPA cyber grand challenge: a competitor's perspective[J]. IEEE Security & Privacy, 2015, 13(6): 72-76.

[3] BRUMLEY D, POOSANKAM P, SONG D, et al. Automatic patch-based exploit generation is possible: techniques and implications[C]//Proceedings of 2008 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2008: 143-157.

[4] AVGERINOS T, CHA S K, HAO B L T, et al. AEG: automatic exploit generation[C]//Network and Distributed System Security Symposium. San Diego: DBLP, 2011: 1-18.

[5] AVGERINOS T, CHA S K, REBERT A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.

[6] CHA S K, AVGERINOS T, REBERT A, et al. Unleashing mayhem on binary code[C]//2012 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2012: 380-394.

[7] WANG M H, SU P R, LI Q, et al. Automatic polymorphic exploit

generation for software vulnerabilities[C]//Security and Privacy in Communication Networks. Cham: Springer, 2013: 216-233.

[8] HUANG S K, HUANG M H, HUANG P Y, et al. CRAX: software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations[C]//Proceedings of 2012 IEEE Sixth International Conference on Software Security and Reliability. Piscataway: IEEE Press, 2012: 78-87.

[9] HE L, CAI Y, HU H, et al. Automatically assessing crashes from heap overflows[C]//Proceedings of 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE Press, 2017: 274-279.

[10] WANG Y, ZHANG C, XIANG X B, et al. Revery: from proof-of-concept to exploitable[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2018: 1914-1927.

[11] WU W, CHEN Y, XU J, et al. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities[C]//Proceedings of the 27th USENIX Security Symposium. Berkeley: USENIX Association, 2018: 781-797.

[12] 方皓, 吴礼发, 吴志勇. 基于符号执行的 Return-to-dl-resolve 利用代码自动生成方法[J]. 计算机科学, 2019, 46(2): 127-132.
FANG H, WU L F, WU Z Y. Automatic return-to-dl-resolve exploit generation method based on symbolic execution[J]. Computer Science, 2019, 46(2): 127-132.

[13] HU H, CHUA Z L, ADRIAN S, et al. Automatic generation of data-oriented exploits[C]//Proceedings of the 24th USENIX Security Symposium. Berkeley: USENIX Association, 2015:177-192.

[14] HU H, SHINDE S, ADRIAN S, et al. Data-oriented programming: on the expressiveness of non-control data attacks[C]//Proceedings of 2016 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2016: 969-986.

[15] ISPOGLOU K K, ALBASSAM B, JAEGER T, et al. Block oriented programming: automating data-only attacks[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2018: 1868-1882.

[16] SCHWARTZ E J, AVGERINOS T, BRUMLEY D, et al. Q: exploit hardening made easy[C]//Proceedings of the 20th USENIX Security Symposium. Berkeley: USENIX Association, 2011: 25.

[17] 黄桦烽, 王嘉捷, 杨轶, 等. 有限资源条件下的软件漏洞自动挖掘与利用[J]. 计算机研究与发展, 2019, 56(11): 2299-2314.
HUANG H F, WANG J J, YANG Y, et al. Automatic software vulnerability discovery and exploit under the limited resource conditions[J]. Journal of Computer Research and Development, 2019, 56(11): 2299-2314.

[作者简介]

**黄桦烽**（1988− ），男，福建永春人，中国科学院软件研究所工程师，主要研究方向为计算机系统安全、漏洞自动挖掘与利用。

**苏璞睿**（1976− ），男，湖北宜昌人，博士，中国科学院软件研究所研究员，主要研究方向为系统安全、恶意代码分析、漏洞挖掘。

**杨轶**（1982− ），男，河南鹤壁人，博士，中国科学院软件研究所副研究员，主要研究方向为计算机系统安全、漏洞挖掘与分析。

**贾相堃**（1990− ），男，河北邯郸人，博士，中国科学院软件研究所副研究员，主要研究方向为系统安全、漏洞挖掘与分析。