**引用格式**:黄晖,陆余良,朱凯龙,等. 一种地址泄露敏感的二进制软件漏洞自动验证方法[J]. 信息对抗技术,2024, 3(2):82-94. [HUANG Hui, LU Yuliang, ZHU Kailong, et al. An automatic address leakage sensitive exploit generation method for vulnerabilities in binary programs[J]. Information Countermeasure Technology, 2024, 3(2):82-94. (in Chinese)]

# 一种地址泄露敏感的二进制软件漏洞自动验证方法

黄 晖[1,2],陆余良[1,2*],朱凯龙[1,2],赵 军[1,2]

(1. 国防科技大学电子对抗学院,安徽合肥 230037; 2. 网络空间安全态势感知与评估安徽省重点实验室,安徽合肥 230037)

**摘 要** 软件漏洞自动验证是分析漏洞可利用性、评估其危害性的重要手段。然而在目标系统开启地址空间布局随机化(address space layout randomization, ASLR)漏洞缓解机制条件下,由于缺乏地址泄露事件的构造能力和有效的漏洞利用载荷运行时重定位方法,当前技术无法生成能有效验证漏洞可利用性的输入样本。为解决上述问题,提出了一种地址泄露敏感的二进制软件漏洞自动验证方法。该方法包含完全地址泄露漏洞状态自动构造和运行时环境无关的漏洞利用会话自动生成 2 个阶段。首先,综合执行状态动态监控、地址泄露样本自动构造、地址泄露导引的模糊测试等技术,自动生成能够蕴含执行目标载荷所需的全部地址泄露事件,并于其后触发漏洞的程序状态。然后,基于该漏洞触发状态,综合漏洞可利用状态构造、漏洞利用模板自动提取、基于载荷运行时动态重定位的漏洞可利用性自动验证等技术,自动构造出能够动态适配于目标系统运行环境的漏洞利用会话,并基于该会话自动完成目标漏洞可利用性分析。基于上述技术实现了 LeakableExp 原型系统,并以该原型系统对 2 个测试程序、14 个 CTF、RHG 竞赛赛题程序和 4 个实际漏洞程序进行了实验分析。实验结果表明,LeakableExp 具备在 ASLR 开启条件下,自动泄露目标系统敏感地址、分析漏洞可利用性的能力。

**关键词** 地址泄露;载荷重定位;漏洞自动验证

**中图分类号** TP 393 **文章编号** 2097-163X(2024)02-0082-13

**文献标志码** A **DOI** 10.12399/j.issn.2097-163x.2024.02.008

# An automatic address leakage sensitive exploit generation method for vulnerabilities in binary programs

HUANG Hui[1,2], LU Yuliang[1,2*], ZHU Kailong[1,2], ZHAO Jun[1,2]

(1. College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China;
2. Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China)

**Abstract** Automatic exploit generation is a critical method in evaluating the exploitability and assessing the severity of software vulnerabilities. However, due to lack of ability in construction of address leakage events and effective runtime relocation on exploit payloads, current methods generally fail in generating exploits adaptable to environments where the address space layout randomization(ASLR) vulnerability mitigation option is turned on. To solve the above problem, an automatic address leakage sensitive exploit generation method was proposed for vulnerabilities in binary programs. This method is composed of 2 stages, one for

**收稿日期**:2023-05-08 **修回日期**:2023-07-17

**通信作者**:陆余良,E-mail:luyuliang@nudt.edu.cn

**基金项目**:国家自然科学基金资助项目(62202484);国防科技大学青年自主创新基金资助项目(ZK23-48)

automatic construction of vulnerable program state under complete address leakage, the other automatic runtime environment irrelevant exploitation session generation. In the first stage, techniques including dynamic execution monitoring, automatic address leakage sample construction and address leakage guided fuzzing were employed to generate vulnerable program state that can not only trigger all the address leakage events necessary to execute the target payload, but also invoke some vulnerability afterwards. In the second stage, those were performed including exploitable state construction, automatic exploitation template extraction and exploit payload runtime relocation based automatic vulnerability exploitability verification on the vulnerable program state generated by stage 1, exploitation session that can dynamically fit on the runtime environment of the target system automatically constructed. The generated session is then used to automatically evaluate the exploitability of the target vulnerabilities. LeakableExp was implemeted based on the aforementioned techniques and was evaluated on 2 test programs, 14 CTF & RHG challenges and 4 real world programs. The results of the experiments demonstrate that LeakableExp is effective in address leakage test case generation and exploitability evaluation for vulnerabilities under ASLR environments.

**Keywords**　address leakage; payload relocation; automatic exploit generation

## 0　引言

漏洞自动验证(automatic exploit generation, AEG)技术是分析软件漏洞可利用性并进一步评估其危害性的有效手段。该技术当前已成为信息安全领域中的一个研究热点，先后出现了 AEG[1]、CRAX[2]、rex[3]、Mayhem[4]等原型方案。在给定能够触发漏洞的输入实例的情况下，这些技术方案一般依托于具体-符号混合执行技术[5]对目标系统执行过程动态追踪。当漏洞触发时，尝试构造漏洞利用条件，从漏洞触发状态转换得到漏洞可利用状态，进而应用基于可满足性模理论(satisfiability module theory, SMT)的约束求解技术[6]自动生成漏洞利用样本。尽管它们能够对栈溢出[7-8]、堆溢出[9-11]、格式化字符串[12-13]等典型二进制软件漏洞进行安全分析，能够绕过数据执行保护[14]、栈保护变量[15]、SafeSEH[16]等漏洞缓解机制，但在目标系统开启地址空间布局随机化(address space layout randomization, ASLR)[17]漏洞缓解机制条件下，当前技术方案普遍无法完成全自动化的漏洞验证分析。

在 ASLR 开启条件下，目标进程地址空间内堆、栈、部分程序模块的加载基址都将发生变化。受到影响的程序模块包括目标程序运行过程中依赖的系统运行时库、第三方程序库等关键组件。目标程序本身如果以位置无关可执行文件(position-independent executable, PIE)[18]方式编译，亦会受到影响。该保护机制目前被广泛部署应用于当前主流操作系统中。安全分析人员对运行于该环境下软件漏洞的可利用性进行人工分析，需要首先从目标进程地址空间内泄露出某些敏感地址，进而基于该地址对后续预期传递给目标程序的 ROP[19]等地址复用型漏洞利用载荷进行手工重定位，得到适配于目标系统运行时环境的漏洞利用载荷，随即将其传递给目标程序。通过观察漏洞利用载荷的预期执行效果是否达到，判断漏洞是否可被利用。当前的技术方案不能适用于 ASLR 开启条件下的漏洞自动验证，根本原因包括以下 2 点：

1) 缺乏地址泄露事件的动态构造方法。在漏洞触发时刻，当前技术往往只关注于从漏洞触发状态到漏洞可利用状态的“直接转化”，忽视了地址泄露这一 ASLR 条件下漏洞验证分析必需的中间过程的动态构建。

2) 载荷重定位功能缺失。当前解决方案仅仅能够生成适配于分析时环境的漏洞利用载荷。然而在 ASLR 开启条件下，由于运行时环境的地址空间布局不同于分析时环境，载荷重定位技术缺失，将导致生成的漏洞利用样本无法动态适配于实际运行环境。

针对上述问题，本文提出了一种地址泄露敏感的二进制软件漏洞自动验证方法。首先，以执

行状态动态监控技术动态记录程序执行过程中已经发生或可能发生的地址泄露事件。随之，依据该上下文信息，或不断迭代计算出具备地址泄露能力且能够触发漏洞的输入实例；或在判定当前程序状态蕴含必需地址泄露事件且可以触发控制劫持漏洞的情况下，生成可适配于分析时环境的漏洞利用模板，并依托基于载荷运行时动态重定位的漏洞可利用性自动验证技术，自动化地以“目标系统运行时环境动态适配”方式，完成漏洞可利用性分析。并由此构建了 LeakableExp 原型系统。对包含 20 个程序的测试程序集，LeakableExp 可以成功为其中的 16 个漏洞程序，在 ASLR 开启条件下，自动完成漏洞可利用性分析验证。

本文的主要贡献如下：

1）提出了一种完全地址泄露漏洞状态自动构造技术。在执行状态监控、漏洞测试用例自动生成过程中创造性地引入对地址泄露事件的动态感知和构造能力，以迭代进化的方式，自动生成能够泄露执行目标载荷所需的全部地址信息，并且能够进一步触发新的漏洞的程序执行状态。

2）提出了一种新颖的运行时环境无关的漏洞利用会话自动生成技术。在漏洞可利用状态构造过程中，动态提取漏洞利用模板信息。随即综合该模板信息对分析时漏洞利用载荷进行面向实际运行环境的动态重定位，实时生成与目标系统运行时环境适配的漏洞利用会话。

3）基于上述技术，形成了地址泄露敏感的二进制软件漏洞自动验证方法，设计实现了 LeakableExp 原型系统，并通过实验验证了该原型系统在 ASLR 保护机制开启条件下程序漏洞可利用性自动分析方面的应用有效性。

## 1 地址泄露敏感的二进制软件漏洞自动验证方法流程

本文构建的地址泄露敏感的二进制软件漏洞自动验证方法工作流程如图 1 所示（其中，蓝色部分为本文的技术贡献），包括完全地址泄露漏洞状态自动构造和运行时环境无关的漏洞利用会话自动生成 2 个阶段。在第 1 阶段首先尝试计算出能够蕴含执行预期地址复用载荷所需的地址泄露事件，且能触发漏洞的完全地址泄露漏洞触发状态；在第 2 阶段，基于该完全地址泄露漏洞触发状态，构建出可以灵活适配于目标程序运行环境的漏洞利用会话。

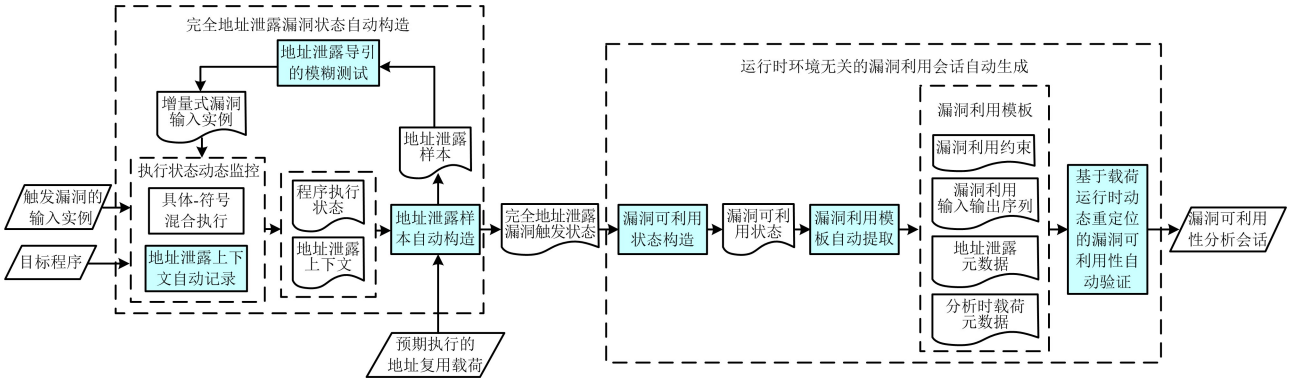


**图 1 地址泄露敏感的二进制软件漏洞自动验证方法流程**

**Fig. 1 Workflow of address leakage sensitive automatic vulnerability exploitation for binary programs**

具体而言，完全地址泄露漏洞状态自动构造阶段首先对程序执行状态动态追踪。在推进程序执行状态不断更新的同时，累积记录对应的地址泄露上下文。地址泄露样本自动构造模块随之基于程序执行状态、地址泄露上下文 2 部分信息，结合预期执行的地址复用载荷，判断该载荷运行需要的地址泄露事件在当前程序状态下是否已触发。在判定所需地址泄露事件未完全触发情况下，该模块尝试构造出在目标程序运行期间能够导致某些敏感地址泄露事件发生的地址泄露样本，驱动地址泄露导引的模糊测试模块进一步生成能够保持该地址泄露事件发生，并于其后触发程序漏洞的增量式漏洞输入实例。该增量式漏洞输入实例将作为新的“触发漏洞输入实例”提供给执行状态动态监控模块，驱动新一轮迭代分析。

当地址泄露样本自动构造模块判定当前程序执行状态蕴含全部预期地址泄露事件时，如果当前程序状态能够进一步触发程序漏洞，该程序状态即被判为完全地址泄露漏洞触发状态，导引

分析进入运行时环境无关的漏洞利用会话自动生成阶段。在这一阶段，首先尝试由完全地址泄露漏洞触发状态转化得到漏洞可利用状态。随之应用漏洞利用模板自动提取技术，从漏洞可利用状态中提取出包含漏洞利用约束、漏洞利用输入输出操作序列、地址泄露元数据、分析时载荷元数据等信息的漏洞利用模板。进而以基于载荷运行时动态重定位的漏洞可利用性自动验证技术基于该模板信息自动生成漏洞可利用性分析会话。该会话以动态适配的方式与目标漏洞运行时环境实时进行数据交互，最终完成 ASLR 开启条件下程序漏洞可利用性的自动分析过程。

## 2　完全地址泄露漏洞状态自动构造

在给定目标程序、触发漏洞的输入实例、预期执行的地址复用载荷条件下，完全地址泄露漏洞状态自动构造阶段综合执行状态动态监控、地址泄露样本自动构造、地址泄露导引的模糊测试等技术，以迭代反馈的方式，不断驱动测试用例生成朝着“能够引发地址泄露事件，同时进一步触发程序漏洞”的方向发展，最终计算得到完全地址泄露漏洞触发状态，导引漏洞验证分析进入第 2 阶段。

### 2.1　执行状态动态监控

执行状态动态监控引擎依托于动态二进制翻译的软件虚拟机[20]，使用具体-符号混合执行技术，沿着能够触发漏洞的输入实例对应的程序执行轨迹对目标程序执行过程动态分析。在此背景下，程序执行状态可以表示为五元组 $S_{\text{state}}$：$\langle \Sigma,\mu,\Delta,\Pi,\delta\rangle$，其中，$\Sigma:\mathbf{Z}\rightarrow$Instructions 表示虚拟地址到程序指令的映射关系；$\mu:\mathbf{Z}\rightarrow\mathbf{Z}\cup S_{\text{SYM}}$（$S_{\text{SYM}}$ 表示符号域 SYMBOLICS）、$\Delta:N_{\text{REG\_NAMES}}\rightarrow\mathbf{Z}\cup S_{\text{SYM}}$ 分别表示各个内存单元、寄存器对应的定值状态（在具体-符号执行背景下，定值 $x$ 可以是整数域内的具体数值，亦可是符号域 SYMBOLICS 内一阶逻辑形式的符号表达式），其中 $\Delta(''\text{pc}'')$ 表示当前程序计数器对应的定值；$\Pi\in S_{\text{SYM}}$ 表示路径约束条件；$\delta:\mathbf{Z}\rightarrow S_{\text{SYM}}$ 记录目标程序在读取外部输入过程中，在对应读取偏移位置引入的符号变元。

随着执行迹中每条指令的顺序执行，指令对应的操作语义不断地对程序执行状态产生影响[21]。为记录程序执行过程中累积发生的地址泄露事件，定义地址泄露上下文为五元组 $L_{\text{leakCtx}}$：$\langle\gamma,\upsilon,M,\sigma,\tau\rangle$，其中，$\gamma$ 表示预期的信息泄露渠道（包括程序的标准输出、网络连接的通信端点等）；$\upsilon$ 表示维护当前累积向预期信息泄露渠道输出的字节数目；$M:N_{\text{MOD\_NAMES}}\rightarrow\{\langle s_{\text{start}},e_{\text{end}}\rangle\mid s_{\text{start}}\in\mathbf{Z},e_{\text{end}}\in\mathbf{Z},s_{\text{start}}<e_{\text{end}}\}$ 表示目标程序运行期间，进程地址空间内栈、堆、各个可执行模块对应的地址映射范围，该部分信息基于操作系统语义动态抽取机制，依据目标进程地址空间实际映射状况动态提取；$\sigma:N_{\text{MOD\_NAMES}}\rightarrow\{\langle a_{\text{addr}},o_{\text{offset}},f_{\text{fmt}}\rangle\mid a_{\text{addr}}\in\mathbf{Z},o_{\text{offset}}\in\mathbf{Z},f_{\text{fmt}}\in\{R_{\text{RAW}},D_{\text{DECIMAL}},H_{\text{HEX}}\}\}$ 用于记录目标程序运行过程中输出缓冲区必定包含敏感地址时产生的确定型地址泄露事件，$\sigma$ 为确定型地址泄露上下文；$\tau$：$\{\langle a_{\text{addr}},s_{\text{sz}},s_{\text{start-off}},f_{\text{fmt}}\rangle\mid a_{\text{addr}}\in S_{\text{SYM}}$ 或 $s_{\text{sz}}\in S_{\text{SYM}},f_{\text{fmt}}\in\{R_{\text{RAW}},D_{\text{DECIMAL}},H_{\text{HEX}}\}\}$ 用于记录程序执行过程中发生输出 API 函数调用时，表示输出缓冲区地址或大小的参数为符号数值的情况。本文定义这类事件为未决型地址泄露事件，$\tau$ 为未决型地址泄露上下文。

本文基于对外部输出相关 API 函数的动态挂钩，实现对上述两型地址泄露事件的动态捕获。以 write 函数为例，当目标程序运行期间发生对该函数的动态调用时，$\Delta(''\text{pc}'')$ 指向 write 函数的运行时地址，对应钩子函数即时触发，分析引擎遂进行如算法 1 所示的分析计算。

**算法 1**　API 函数 write($f_{\text{fd}},b_{\text{buf}},s_{\text{sz}}$)调用位置地址泄露上下文收集算法

输入：当前程序执行状态 $S_{\text{state}}$：$\langle\Sigma,\mu,\Delta,\Pi,\delta\rangle$ 及对应地址泄露上下文 $L_{\text{leakCtx}}$：$\langle\gamma,\upsilon,M,\sigma,\tau\rangle$。$\Delta(''\text{pc}'')$ 当前指向 write 函数的入口地址，$f_{\text{fd}},b_{\text{buf}},s_{\text{sz}}$ 为当前 write 函数调用对应的参数。

输出：write($f_{\text{fd}},b_{\text{buf}},s_{\text{sz}}$)函数执行结束后的程序执行状态 $S_{\text{state}'}$ 和地址泄露上下文 $L_{\text{leakCtx}'}$。

```
1  S_state' = S_state
2  L_leakCtx' = L_leakCtx
3  if (source(f_fd) ∈ γ) then
4     if (b_buf ∈ S_SYM or s_sz ∈ S_SYM) then
5        S_state' = ⟨Σ, μ, Δ["eax"→s_sz], Π, δ⟩
6        L_leakCtx' = ⟨γ, υ + s_sz, M, σ, τ[⟨b_buf, s_sz, υ,
         R_RAW⟩]⟩
7     else
8        for o_offset = 0 to s_sz:
9           a_addr = μ(b_buf + o_offset)
10             for m in M:
```

11 if ($M[m].a_{addr}<a_{addr}$ and $M[m].a_{addr}+M[m].o_{offset}\geqslant a_{addr}$)

12 tuple=$\langle a_{addr}-M[m].a_{addr},\upsilon+o_{offset},R_{RAW}\rangle$

13 $\upsilon=\upsilon+s_{sz}$

14 $\sigma=\sigma[m\rightarrow tuple]$

$L_{leakCtx'}=\langle a_{addr}-M[m].s_{start},\upsilon,M,\sigma,\tau\rangle$

return $\langle S_{state'},L_{leakCtx'}\rangle$

在确定文件描述符参数 $f_{fd}$ 指向预期的信息泄露渠道的情况下(第 3 行),分析引擎检查 $b_{buf}$、$s_{sz}$ 参数是否为可以受到外部输入影响的符号数值,并在确定的情况下,以$\langle b_{buf},s_{sz},\upsilon,R_{RAW}\rangle$更新未决型地址泄露上下文(第 4～6 行)。该未决型地址泄露上下文随后可以按需转化为确定型地址泄露事件。否则,即在当前状态存储映射 $\mu$ 中,检查 $\mu[b_{buf}\ \cdots\ b_{buf}+s_{sz}]$区域内是否包含位于某个当前已加载模块内的程序地址(第 8～14 行)。若确定,即可判定当前函数调用存在着确定型地址泄露事件,遂在确定型地址泄露上下文中添加 $m\rightarrow\langle a_{addr}-M[m].a_{addr},\upsilon+o_{offset},R_{RAW}\rangle$映射,实现该地址泄露事件的实时记录。

## 2.2 地址泄露样本自动构造

执行状态监控模块每推演得到一个新的程序状态 $S_{state}$:$\langle\Sigma,\mu,\Delta,\Pi,\delta\rangle$及相应的地址泄露上下文 $L_{leakCtx}$:$\langle\gamma,\upsilon,M,\sigma,\tau\rangle$,地址泄露样本自动构造模块即刻参考预期执行的地址复用型功能载荷所位模块信息,区分计算以下几种情况:

1) 如果 $\sigma$ 中包含目标功能载荷所位模块相关记录,且状态 $S_{state}$ 中维护的路径约束条件 $\Pi$ 能够与某些程序漏洞的触发条件同时满足,即确认当前漏洞触发时,执行目标载荷所需的地址泄露事件皆已完成,状态 $S_{state}$ 即被判为完全地址泄露漏洞触发状态,并作为输入驱动系统进入第 2 阶段安全分析。

2) 如果 $\sigma$ 中不包含目标功能载荷所位模块相关记录,即尝试综合 $S_{state}$、$L_{leakCtx}$ 两部分信息,转化得到“能够导引程序触发地址泄露事件且不会触发崩溃”的地址泄露状态。基于转化得到的地址泄露状态,应用约束求解技术自动计算得到地址泄露样本,随即终止该地址泄露状态。计算得到的地址泄露样本将提供给本阶段随后地址泄露导引的模糊测试部件,驱动后者的测试用例生成过程更进一步逼近完全地址泄露漏洞触发状态。

3) 如果上述 2 种情况都不成立,则任由执行状态监控模块继续基于程序执行状态 $S_{state}$ 推演分析得到新的程序状态,继而重复上述计算过程。

地址泄露状态自动构造是本模块的核心工作。本文提出以下 3 种计算方案。

### 2.2.1 覆盖条件转化

图 2 给出了该方法的一个计算样例。图 2(a)所示程序包含一个栈溢出漏洞。图 2(b)给出了 func 函数被调用时程序的栈空间布局状态。对这个程序,漏洞触发样本一般能够导引程序产生图 2(c)所示的栈空间数据状态。此时栈上原本保存的返回地址(可以在图 2(b)时刻,对程序执行状态分析确定)被直接覆盖,然而由于缺乏地址泄露信息,分析引擎无法直接完成漏洞自动利用过程。对于这种情况,覆盖条件转化方法通过修改载荷长度,精确控制 buffer 缓冲区覆盖字节数为 0x60,将图 2(c)所示的栈数据布局状态转为图 2(d)所示的栈数据布局状态。由此即可将原本栈溢出漏洞触发事件转换为地址泄露事件,藉由程序第 8 行包含的 printf 输出函数,实现栈上保存的栈地址(EBP)、func 函数调用点返回地址等信息的自动泄露。

### 2.2.2 地址泄露载荷布置

在当前程序状态能够导致控制流劫持的情况下,如果当前程序状态内存在某个内存区域$[a_{addr},a_{addr}+s_{sz}]$可以用于布局“能够泄露模块 $m$ 内某个敏感地址 addr_leak”的利用载荷,并且可以将控制流导引到该利用载荷上,分析引擎计算出对应的控制流导引约束 $c_{cflow\_constraint}$ 和载荷布局约束 $c_{leak\_constraint}$,并将这两部分约束条件与当前程序状态路径约束 $\Pi$ 合取。在合取得到的约束条件为可满足情况下,程序状态$\langle\Sigma,\mu,\Delta,\Pi\wedge c_{cflow_constraint}\wedge c_{leak_constraint},\delta\rangle$即为可以在运行时泄露出模块 $m$ 内地址信息 addr_leak 的地址泄露状态。

图 3 给出了该方法的一个计算样例。对于图 3(a)所示程序 ,图 3(b)所示的漏洞触发状态可以导致程序执行崩溃,但其间并不涉及对输出函数的调用,因此无法产生信息输出。然而,我们可以从主程序模块中提取出能够从全局偏移表(Global Offset Table, GOT)[22]中泄露出 libc.so 程序库内 puts 函数运行时地址的 ROP 序列,将

该 ROP 序列布置于漏洞触发状态中相应的可容纳执行载荷的内存区域，并于其后补充 main 函数的地址信息，构造出栈布局如图 3(c)所示的程序执行状态。该程序执行状态可以泄露出 libc.so 中某个 API 函数的地址，同时驱动程序产生新一轮数据读取操作。

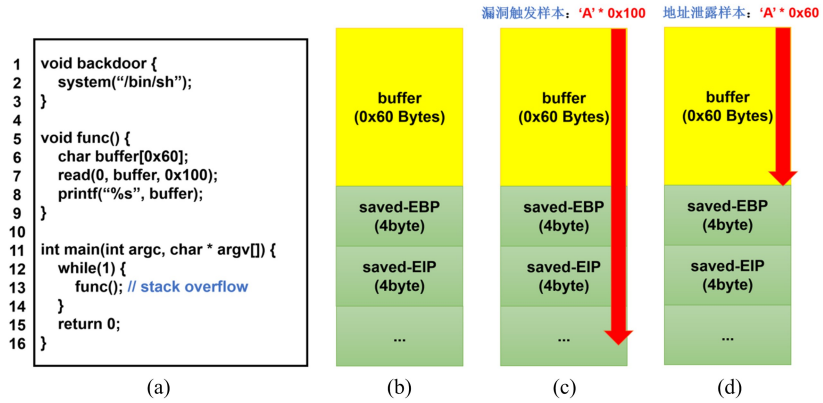


**图 2　基于覆盖条件转化的地址泄露状态构造示例**

**Fig. 2　Sample of address leakage state construction through transformation on overwrite condition**

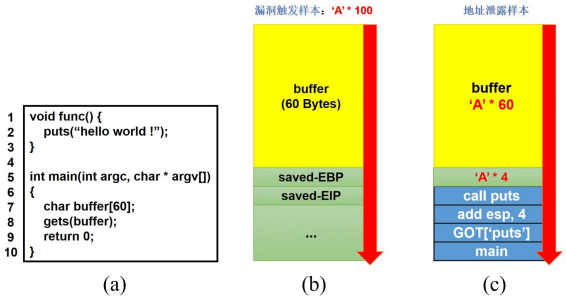


**图 3　基于载荷布置的地址泄露状态构造示例**

**Fig. 3　Sample of address leakage state construction through payload arrangement**

### 2.2.3　未决型地址泄露事件动态转换

同样在当前程序状态能够导致控制流劫持的情况下，如果当前未决型地址泄露上下文 $\tau$ 内记录的某个条目 $\langle b_{\text{buf\_sym}}, s_{\text{sz\_sym}}\rangle$ 可以被改造成关于模块 $m$ 的确定型地址泄露事件，分析引擎综合当前路径约束 $\Pi$ 和完成该转换所需构造的数据约束，合取形成新的路径约束条件，由此完成地址泄露状态的构造。

图 4 给出了该方法的一个计算样例。该程序第 10 行 printf 函数调用的输出缓冲区可以被外部输入控制。如 2.1 节所述，在执行状态动态监控过程中将形成一个未决型地址泄露上下文。当执行监控追踪程序执行过程到第 11 行，触发对应的栈溢出漏洞时，当前并无任何确定型地址泄露事件发生。然而，此时可以使用约束求解技术，将上述未决型地址泄露上下文中记录的符号地址操作数 data+index 指向包含敏感地址操作数的内存区域（譬如全局偏移表）。由此将未决型地址泄露事件转化成确定型地址泄露事件，满足目标载荷应用的地址泄露需求。

```
1 #include <stdio.h>
2
3 char data[1024];
4
5 int main(int argc, char * argv[])
6 {
7     char buffer[10];
8     read(0, buffer, 10);
9     int index = *((int *)(buffer));
10    printf("content: %d", data[index]);
11    read(0, buffer, 1024);
12    return 0;
13 }
```

**图 4　未决型地址泄露样例**

**Fig. 4　Example of pending address leakage**

在应用上述方法计算得到地址泄露状态后，分析引擎随之将该地址泄露状态作为输入，应用算法 2 所示的地址泄露样本自动生成算法，对路径约束 $\Pi$ 进行约束求解（第 2 行），并根据输入变元序列 $\delta$ 进行相应截断（第 3～5 行），自动计算出能够导引执行到该程序状态的地址泄露样本。在完成地址泄露样本计算后，原始程序状态和构造出的地址泄露状态被即时销毁。分析引擎即时将该地址泄露样本提供给 2.3 节所述的地址泄露导引的模糊测试技术，驱动后者进一步生成能够引发地址泄露样本所蕴含地址泄露事件，同时更进一步触发新的程序漏洞的输入实例。

**算法 2**　地址泄露样本自动生成算法

输入：地址泄露状态 $S_{\text{state}}$：$\langle \Sigma, \mu, \Delta, \Pi, \delta\rangle$

输出：地址泄露样本内容

```
solver = SMTSolver()
content = solver.solve(state.Π)
out_content = new byte[state.δ.count()]
for index in state.δ:
    out_content[index] = content[index]
return out_content
```

### 2.3 地址泄露导引的模糊测试

地址泄露导引的模糊测试旨在基于 2.2 节计算得到的地址泄露样本，进一步构造得到增量式漏洞输入实例。该部分技术如图 5 所示。类同于基于覆盖的模糊测试[23-24]，本文的地址泄露导引的模糊测试技术维护着一个全局种子队列，每轮迭代生成测试用例时，都会从该队列中选取一个种子进行变异生成新的测试用例。种子队列的数据来源包括 2 种，即 2.2 节中地址泄露样本构造产生的地址泄露样本和每轮迭代过程生成的测试用例。仅当这些测试用例可以被评估为“有新贡献的种子输入”，才会将其加入种子队列。

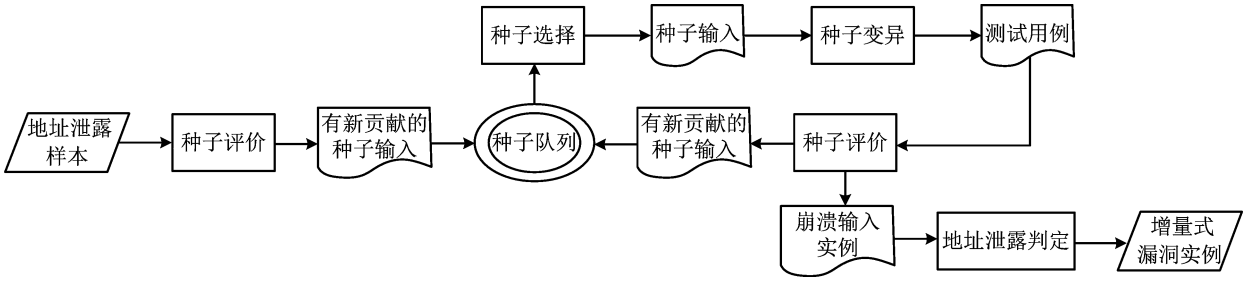


**图 5 地址泄露导引的模糊测试技术框架图**

**Fig. 5 Framework of address leakage guided fuzzing**

然而，与基于覆盖的模糊测试不同的是，本文将 2.1 节所述执行状态监控过程中的地址泄露上下文记录机制引入到种子评价阶段，进而对“有新贡献的种子输入”的定义标准进行拓展：如果一个测试用例能够产生新的边覆盖情况，并且能够在执行过程中动态泄露出进程地址空间内某些模块内部地址信息，即认为该测试用例为“能够产生新贡献的种子输入”。通过确保参与变异的种子实例都能够触发地址泄露事件，提升生成的测试用例蕴含地址泄露事件的概率。同时对生成的崩溃输入实例进行了额外的执行监控，仅当判定其执行过程中能够产生某些地址泄露事件，才会将该崩溃输入实例判定为增量式漏洞实例，提供给执行状态动态监控模块进行新一轮迭代分析。由此导引分析持续朝向能够触发地址泄露事件，并于随后触发漏洞的方向发展，以不断进化的方式驱动完全地址泄露漏洞触发状态的计算获取。

## 3 运行时环境无关的漏洞利用会话自动生成

运行时环境无关的漏洞利用会话自动生成是漏洞验证分析的第 2 阶段。该阶段首先尝试将前一阶段输出的完全地址泄露漏洞触发状态转化成漏洞可利用状态，从中提取出漏洞利用模板。随即以基于载荷运行时动态重定位的漏洞可利用性自动验证技术基于该模板，生成能够动态适配于目标系统运行时环境的漏洞利用会话，自动化地完成针对目标漏洞的可利用性分析。

### 3.1 漏洞可利用状态构造

漏洞可利用状态构造过程如下。对于前一阶段计算得到的完全地址泄露漏洞触发状态 $S_{\text{state}}=\langle \Sigma,\mu,\Delta,\Pi,\delta\rangle$，分析引擎首先确定能够劫持程序控制流到外部输入所在内存区域的漏洞利用原语，计算得到布局该原语对应的约束条件 $c_{\text{primConstraint}}$。随即基于应用该原语产生的中间状态 $\langle \Sigma,\mu',\Delta',\Pi',\delta\rangle$，依次尝试应用 return to libc、return to dl-resolve 等漏洞利用技术[25]计算得到各种可行载荷布局方案对应的约束条件。如果某些载荷布局约束条件 $c_{\text{expConstraint}}$ 与原语布局约束条件 $c_{\text{primConstraint}}$、路径约束条件 $\Pi$ 可以同时满足，这 3 部分约束的合取即形成漏洞利用条件。程序状态 $\langle \Sigma,\mu,\Delta,\Pi\wedge c_{\text{prim\_constraint}}\wedge c_{\text{exp\_constraint}},\delta\rangle$ 即为能够蕴含执行目标载荷所需全部地址泄露事件的漏洞可利用状态。

### 3.2 漏洞利用模板自动提取

漏洞利用模板自动提取模块旨在从 3.1 节构造得到的漏洞可利用状态中，自动提取出包含漏洞利用约束条件、漏洞利用 IO 序列、地址泄露元数据、分析时载荷元数据等四部分信息的漏洞利

用模板。其中漏洞利用约束条件已在 3.1 节完成计算。这里将被转储为 SMT-LIB2[26] 形式的离线约束，便于 3.3 节通过约束覆写进行数据重算。

漏洞利用 IO 序列指明目标程序从程序开始运行到漏洞成功利用的完整过程中，输入输出事件发生的具体次序。其中从程序启动到漏洞触发阶段的 IO 序列信息已由执行状态动态监控完成收集，从漏洞触发到漏洞利用阶段的 IO 序列信息则在漏洞利用约束构建阶段同步计算得到。这两部分信息进行拼接形成的完整序列，将辅助后端技术重现完成漏洞利用所必需的信息交互过程。

地址泄露元数据信息为漏洞利用状态关联的地址泄露上下文中记录的每个确定型地址泄露事件，维护对应的五元组记录 $A_{\text{addrleakMd}}$：$\langle m_{\text{modName}}, o_{\text{mod}}, o_{\text{leak}}, f_{\text{leakFmt}}, l_{\text{leakLen}} \rangle$。其中，$m_{\text{modName}} \in N_{\text{MOD\_NAMES}}$ 表示泄露敏感地址所位模块名称；$o_{\text{mod}} \in \mathbf{Z}$ 表示敏感地址在所位模块中的偏移；$o_{\text{leak}} \in \mathbf{Z}$、$f_{\text{leakFmt}} \in \{R_{\text{RAW}}, D_{\text{DECIMAL}}, H_{\text{HEX}}\}$、$l_{\text{leakLen}} \in \mathbf{Z}$ 分别表示敏感地址在输出数据流中的起始偏移位置、输出格式和输出内容所占字节数目。这些信息除 $l_{\text{leakLen}}$ 外，都可以从地址泄露上下文中直接提取。$l_{\text{leakLen}}$ 则可综合敏感地址所位模块基址（由地址泄露上下文维护的地址映射信息 $M$ 获取）、$o_{\text{mod}}$、$f_{\text{leakFmt}}$ 等信息计算得到。这些信息可为漏洞利用会话在与目标程序的实时交互中，从输出数据流中动态提取泄露地址信息提供支持。

分析时载荷元数据信息 $p_{\text{payloadMds}}: N_{\text{MOD\_NAMES}} \rightarrow 2^{\{\langle m_{\text{modOff}}, a_{\text{addrAna}} \rangle \mid m_{\text{modOff}} \in \mathbf{Z}, a_{\text{addrAna}} \in \mathbf{Z}\}}$ 为漏洞利用约束条件生成过程中使用到的每个功能载荷，以功能载荷所位模块为单位，记录载荷在所位模块内的偏移、载荷在分析时环境内的虚拟地址两部分信息。这些信息在 3.1 节漏洞可利用状态构造期间，随漏洞利用载荷布局同步计算提取。该部分信息将为对应载荷数据在程序运行期间动态重定位提供支撑。

### 3.3　基于载荷运行时动态重定位的漏洞可利用性自动验证

基于 3.2 节计算得到的漏洞利用模板，分析引擎对 ASLR 开启条件下运行的目标程序，进行自动化的漏洞可利用性验证。在以漏洞利用 IO 序列规范的次序进行数据交互过程中，依据地址泄露元数据信息从程序的输出流中动态提取出泄露的敏感地址，并以此结合漏洞利用约束条件、分析时载荷元数据信息对功能载荷进行实时重定位。随即将该重定位后的功能载荷发送给目标程序，通过观察预期的利用效果是否实现，确定目标漏洞的可利用性。具体过程如算法 3 所示。

**算法 3**　基于载荷运行时动态重定位的漏洞可利用性自动验证算法

输入：在 ASLR 开启环境中运行的目标进程 $P$，漏洞利用约束条件 $c_{\text{smtConstraint}}$，漏洞利用 IO 序列 $I_{\text{IOSeqs}}$，地址泄露元数据信息 $A_{\text{AddrLeakMds}}$，分析时载荷元数据信息 $p_{\text{payloadMds}}$。

输出：漏洞是否在 ASLR 开启条件下可被利用。TRUE 表示可以，FALSE 则表示本次会话不能确定。

```
1  proc = P.connect()
2  solver = SMTSolver()
3  content = solver.solve(c_smtConstraint)
4  read_offset = 0
5  write_offset = 0
6  for each I_IOEntry in I_IOSeqs:
7      if I_IOEntry.type == READ then
8          proc.send(content + read_offset, I_IOEntry.size)
9          read_offset = read_offset + I_IOEntry.size
10     else if I_IOEntry.type == WRITE then
11         (recvContent, recvLen) = proc.recv()
12         leak_entries = A_AddrLeakMds.get_relevant(write_offset, recvLen)
13         foreach entry in leak_entries:
14             (has_leak, leak_addr) = extract_leakaddr(recvContent, write_offset, recvLen,
15                 entry.leakOff, entry.leakLen, entry.leakFmt)
16             if (! has_leak) then
17                 write_offset = write_offset + recvLen
18                 break
19             leak_module = entry.modName
20             leak_modbase = leak_addr - entry.modOff
21             for payload in p_payloadMds[leak_module]:
22                 payload_runtime_addr = payload.modOff + leak_modbase
23                 c_smtConstraint = rewrite(c_smtConstraint, payload.analyisis_addr,
24 payload_runtime_addr)
25         content = solver.solve(c_smtConstraint)
```

```
26          write_offset = write_offset + recvLen
27    if (proc. send _ CMD _ and _ execute ( ) = = OK) then
28          return TRUE
29    else
30          return FALSE
```

算法 3 首先与目标进程 $P$ 建立连接(第 1 行),以 $c_{\text{smtConstraint}}$ 漏洞利用条件调用约束求解器计算得到将在第 1 次数据传输中传递给目标程序的会话数据 content。随即依据 $I_{\text{IOSeqs}}$ 中指定的 IO 操作顺序,完成漏洞利用所需的数据交互过程。区分 2 种情况完成对 IO 操作的处理计算:

1) 若当前 IO 操作 $I_{\text{IOEntry}}$ 为 READ 类型(此时表示目标进程正等待接收输入),即从当前会话数据 content 中,偏移 read_offset 处开始读取 $I_{\text{IOEntry}}$. size 长度的内容,再传递给目标进程。在完成上述操作后,对输入数据流偏移 read_offset 进行相应更新(第 7～9 行)。

2) 若当前 IO 操作 $I_{\text{IOEntry}}$ 为 WRITE 类型(此时表示目标进程刚刚完成数据输出),即在完成数据接收的同时(第 11 行),检查该部分输出数据中是否包含地址泄露信息(第 12 行)。在确定包含的情况下,按照 $A_{\text{AddrLeakMds}}$ 中对应地址泄露元数据项目中指明的格式信息,从该输出数据流中提取出泄露的敏感地址(第 14～15 行),进而动态计算出该敏感地址所在模块的运行时加载地址(第 20 行)。随即基于该地址信息,计算出 $c_{\text{smtConstraint}}$ 中使用到的该模块内所有数据载荷对应的运行时地址,并以该运行时地址重写 $c_{\text{smtConstraint}}$(第 21～24 行)。该重写过程事实上通过约束条件的更新,隐式完成分析时载荷向运行时载荷的动态重定位。基于重写得到的新的约束条件,分析引擎随即应用约束求解技术计算出新的会话数据 content(第 25 行)。重定位过程保证了该数据中蕴含的泄露地址相关载荷部分在随后新一轮的数据读取会话中,作为输入数据提供给目标程序时,能够适配于目标进程的运行时地址空间布局,进而得到有效执行。

当 $I_{\text{IOSeqs}}$ 中指定的 IO 操作过程全部完成后,分析引擎发送特定命令给目标程序(第 27～30 行)。如果该命令能够正确执行并反馈输出到发送端,即可判定目标漏洞在 ASLR 开启条件下为可利用的。

## 4 原型系统设计与实现

本文基于二进制分析平台 angr[27]、二进制程序模糊测试工具 AFL[28]、二进制程序全系统动态符号执行平台 S2E[29]、漏洞利用开发库 pwntools[30]、Z3 约束求解器[31]等开源项目,实现了 LeakableExp 原型系统,包括 15 334 行 C++代码(S2E 修改部分)、3 471 行 C 代码(AFL 修改部分)、3 173 行 Python 代码(含通过基于 pwntools、Z3 开发形成的基于载荷运行时动态重定位的漏洞可利用性自动验证模块和系统的主体框架两部分内容)。

完全地址泄露漏洞状态自动构造和运行时环境无关的漏洞利用会话自动生成是 LeakableExp 原型系统中的 2 个核心部件。对于前者,本文基于 S2E 平台对目标程序执行状态动态监控,其间通过编写分析插件,完成 API 函数动态挂钩、地址泄露上下文动态记录、地址泄露样本自动构造等核心功能;本文使用 AFL 的 qemu 工作模式对目标程序进行模糊测试。通过修改 AFL 源码,将地址泄露状态作为新的反馈机制引入 AFL 中,由此实现了地址泄露导引的模糊测试功能。

对于运行时环境无关的漏洞利用会话自动生成部件,本文基于 S2E 平台定制插件,实现漏洞可利用状态自动构造、漏洞利用模板自动提取等功能。基于 pwntools、Z3 等开源项目,完成了漏洞利用模板自动解析、运行时泄露地址自动提取、漏洞利用载荷运行时动态重定位等功能,在此基础上形成了基于载荷运行时动态重定位的漏洞可利用性自动验证模块。

## 5 实验分析

本文构建了包含 2 个测试程序、5 个 RHG 竞赛赛题、9 个 CTF 竞赛赛题、4 个含已公开漏洞的真实软件在内的测试程序集,对 LeakableExp 在 ASLR 开启条件下漏洞可利用性自动分析方面的应用有效性进行实验分析。测试程序详细信息见表 1 所列。其中“漏洞缓解机制开启情况”一列中,NX 表示数据执行保护,CANARY 表示栈保护变量。本文将这些程序公布于 https://gitee.com/binooda/leakable-exp-exps.git 上。对于这些程序,分析系统统一在给定能够触发程序内蕴含漏洞、但不能触发预期地址泄露事件的漏洞输入实例的情况下,对漏洞可利用性进行自动分析。实验环境为 1 台安装了 64 位 ubuntu 22.04 版本操作系统的移动工作站,内存容量 64 GB,CPU 型号为 Intel(R) i9-9880H。

**表 1　测试程序集**

**Tab. 1　Benchmark programs for experiments**

| 序号 | 程序名称 | 程序来源 | 漏洞类型 | 主程序以 PIE 方式编译 | 漏洞缓解机制开启情况 | 利用载荷所处模块 |
|---|---|---|---|---|---|---|
| 1 | test1 | 测试程序 | 栈溢出<br>任意地址内存读取 | 是 | ASLR＋NX | libc. so |
| 2 | test2 | 测试程序 | 格式化字符串 | 是 | ASLR＋NX＋CANARY | libc. so |
| 3 | prob1 | RHG 2021 | 栈溢出 | 是 | ASLR＋NX | prob1 |
| 4 | prob11 | RHG 2021 | 栈溢出 | 是 | ASLR | libc. so |
| 5 | question5 | RHG 2021 ljb | 栈溢出 | 是 | ASLR＋NX | question5 |
| 6 | question8 | RHG 2021 ljb | 格式化字符串 | 是 | ASLR | question8 |
| 7 | question13 | RHG 2021 ljb | 栈溢出 | 是 | ASLR | question13 |
| 8 | ret2libc3 | bamboofox | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 9 | ropasaurusrex | PlaidCTF 2013 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 10 | start | pwnable. tw | 栈溢出 | 否 | ASLR | 栈 |
| 11 | pwn-200 | XDCTF 2015 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 12 | no_canary | angstromCTF 2020 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 13 | secret_of_my_heart | pwnable. tw | 堆溢出 | 是 | ASLR＋NX＋CANARY | libc. so |
| 14 | hacknote | pwnable. tw | 释放重引用 | 否 | ASLR＋NX＋CANARY | libc. so |
| 15 | babystack | pwnable. tw | 栈溢出 | 是 | ASLR＋NX＋CANARY | libc. so |
| 16 | zero_task | 0CTF 2019 | 条件竞争、释放重引用 | 是 | ASLR＋NX＋CANARY | libc. so |
| 17 | glFTPd 1. 32 | OSVDB-ID-16373 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 18 | wget 1. 19. 1 | CVE-2017-13089 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 19 | proftpd-1. 3. 0a | CVE-2006-6563 | 栈溢出 | 否 | ASLR＋NX | libc. so |
| 20 | dnsmasq 2. 77 | CVE-2017-14993<br>CVE-2017-14494 | 栈溢出<br>内存读访问越界 | 否 | ASLR＋NX | libc. so |

上述程序统一部署于 ASLR 漏洞缓解机制开启的目标环境内。大部分程序的漏洞可利用性验证，都遵循该条件下漏洞利用验证分析的一般过程：即首先通过以精心构造的输入数据传输给目标程序，诱导目标程序在输出数据流中泄露利用载荷所在程序模块内的某个敏感地址，随之基于该地址信息对利用载荷进行动态重定位，最终将重定位后的利用载荷传输给目标程序，观察预期漏洞利用效果是否触发。某些程序内仅包含 1 个程序漏洞，漏洞可利用分析在会话期间需要对该漏洞多次利用（通过前若干次漏洞利用完成敏感地址泄露，借助最后一次漏洞利用完成重定位后利用载荷的布置执行）以达到利用效果。

但是 question5 和 question13 是 2 个特例。question5 程序在运行期间会主动泄露其主程序模块内偏移 0x8c2 处对应的运行时虚拟地址，分析引擎对该道程序不需要构造地址泄露事件，但需从输出数据流中自动提取出泄露地址，并完成后续可利用性分析过程；question13 的利用载荷与栈溢出漏洞覆盖的程序地址位于同一模块内，分析引擎可以通过该漏洞，直接以利用载荷地址低位字节内容覆盖目标返回地址低位字节的方式完成漏洞可利用性验证分析。

对于上述程序，本文尝试应用 LeakableExp，根据利用载荷所处模块，进行有针对性的地址泄露事件自动构建，同时进一步在开启 ASLR 缓解机制条件下的目标环境中，对漏洞可利用性进行自动判定。实验结果见表 2 所列。

由表 2 可见，在除 question5、question13 外的 18 个需要构造地址泄露事件的测试程序中，LeakableExp 成功地为 15 个构造了地址泄露事件，地址泄露成功率达到 83. 3%；成功地对全部 20 个测试程序中的 16 个完成了漏洞可利用性自

动验证，成功率达到 80%。对于程序本身直接泄露敏感地址的 question5，LeakableExp 亦可动态从输出数据流中识别出敏感地址，完成 ASLR 开启条件下的漏洞可利用性分析。

**表 2 可利用性分析实验结果**

**Tab. 2 Experimental results of vulnerability exploitability analysis**

| 程序名称 | 可利用性验证 | 泄露地址 | 地址泄露构造途径 | 泄露地址信息 |
|---|---|---|---|---|
| test1 | √ | √ | 未决型地址泄露的动态转换 | libc. so 内 API 函数地址 |
| test2 | √ | √ | 未决型地址泄露的动态转换 | 栈地址<br>libc. so 内 API 函数地址 |
| prob1 | √ | √ | 漏洞触发条件转化 | prob1 内偏移 0x9b1 |
| prob11 | √ | √ | 地址泄露载荷布置 | prob11 内偏移 0x855<br>libc. so 内 API 函数地址 |
| question5 | √ | 无须构造<br>（程序直接泄露） | — | question5 内偏移 0x8c2 |
| question8 | √ | √ | 地址泄露载荷布置 | 栈地址<br>question8 内偏移 0x92d |
| question13 | √ | 无须构造<br>（低位地址覆写） | — | — |
| ret2libc3 | √ | √ | 地址泄露载荷布置 | libc. so 内 API 函数地址 |
| ropasaurusrex | √ | √ | 地址泄露载荷布置 | libc. so 内 API 函数地址 |
| start | √ | √ | 漏洞触发条件转化 | 栈地址 |
| pwn-200 | √ | √ | 地址泄露载荷布置 | libc. so 内 API 函数地址 |
| no_canary | √ | √ | 地址泄露载荷布置 | libc. so 内 API 函数地址 |
| secret_of_my_heart | × | × | — | — |
| hacknote | × | × | — | — |
| babystack | × | × | — | — |
| zero_task | × | × | — | — |
| glFTPd 1. 32 | √ | √ | 地址泄露载荷布置 | libc. so 内 API 地址 |
| wget 1. 19. 1 | √ | √ | 地址泄露载荷布置 | libc. so 内 API 地址 |
| proftpd-1. 3. 0a | √ | √ | 地址泄露载荷布置 | libc. so 内 API 地址 |
| dnsmasq 2. 77 | √ | √ | 漏洞触发条件转化 | libc. so 内 API 地址 |

通过对成功的案例分析，认为 LeakableExp 在这些案例上取得较好分析效果的原因在于：

1）LeakableExp 采取的“递进铺设能够产生地址泄露事件并触发新漏洞的程序状态，最终基于完全地址泄露漏洞状态推演可利用状态”的迭代计算模型，能够蕴含对这些案例中的程序漏洞手工利用过程中的各个地址泄露相关的中间步骤；

2）LeakableExp 提出的基于运行时载荷动态重定位的漏洞可利用性自动验证技术可以从漏洞利用分析会话的输出数据流中精确提取泄露的敏感地址，实时完成后续传输的漏洞利用载荷与目标程序运行时环境的动态适配。

本文进一步检查了无法构建所需地址泄漏事件且不能完成漏洞自动验证的 4 个案例程序，分析了失败的原因，主要有以下 3 个方面：

1）babystack 程序包含一个栈溢出漏洞。特殊之处在于该程序运行时开启的漏洞缓解机制包含栈保护变量。这种情况下的漏洞利用，需要首先泄露出栈上存储的栈保护变量，同时在随后地址泄露事件构造及漏洞利用载荷布局阶段，对栈保护变量可能出现的位置，进行有针对性的数据布置和动态重定位。当前系统实现关注于地址泄露事件构造和载荷运行时动态重定位，并未对这种情况的漏洞自动验证过程进行分析。后期可以基于本文所提技术框架，通过补充栈保护变量的泄露手段分析建模相关内容，形成相应的解决方案。

2）zero_task 程序需要使用条件竞争对释放重引用漏洞进行利用，从而达到地址泄露效果。当前系统基于具体-符号混合执行技术对程序执行状态分析推演，然而该技术本身遵循串行计算

模型，并不适用于对并发程序的分析计算。因此对于产生于并发计算环境中的条件竞争漏洞，不能有效验证其可利用性。

3) secret_of_my_heart、hacknote 2 道程序中包含的漏洞分别是 1 个单字节堆溢出漏洞和 1 个释放重引用漏洞。对这 2 种类型漏洞的利用，需要在理清程序内包含所有堆操作原语的基础上，自动化生成能够产生特定内存状态布局的输入实例。系统当前在该方面缺乏有效手段。目前已有相关工作进行初步研究[32-33]，这也是本文后续的一个研究方向。

## 6　结束语

对部署于 ASLR 开启条件下目标环境中的二进制软件漏洞进行可利用性自动验证，是当前漏洞自动验证领域的一个难点问题。本文提出了一种地址泄露敏感的自动解决方案 LeakableExp。通过对地址泄露事件的动态感知和自动构造，将地址泄露能力有效蕴含于生成的输入实例中；在对目标漏洞进行可利用性分析时引入载荷自动重定位机制，使分析时提取的漏洞利用模板能够动态适配于运行时环境。实验结果表明，LeakableExp 不仅可以自动生成具有地址泄露能力的测试用例，还可以自动构造适应于运行时环境的漏洞利用会话，可以有效应用于 ASLR 开启条件下二进制软件漏洞的可利用性自动验证。

## 参 考 文 献


[1] AVGERINOST, CHA S K, HAO B L T, et al. AEG: automatic exploit generation [J]. Communications of the ACM, 2014, 57(2): 74-84.

[2] 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 International Conference on Software Security and Reliability. [S. l.: s. n.], 2012: 78-87.

[3] Rex[EB/OL]. (2007-10-09) [2023-04-14]. https://github.com/angr/rex.

[4] CHA S K, AVGERINOS T, REBERT A, et al. Unleashing mayhem on binary code[C]//Proceedings of 2012 IEEE Symposium on Security and Privacy. [S. l.: s. n.], 2012: 380-394.

[5] YUN I, LEE S, XU M, et al. QSYM: a practical Concolic execution engine tailored for hybrid fuzzing [C]//Proceedings of the 27th USENIX Security Symposium. [S. l.: s. n.], 2018: 745-761.

[6] DE MOURA L, BJORNER N. Satisfiability modulo theories: introduction and applications[J]. Communications of the ACM, 2011, 54(9): 69-77.

[7] 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 Hao, WU Lifa, WU Zhiyong. Automatic Return-to-dl-resolve exploit generation method based on symbolic execution[J]. Computer Science, 2019, 46(2): 127-132. (in Chinese)

[8] 彭建山，奚琪，王清贤. 二进制程序整型溢出漏洞的自动验证方法[J]. 信息网络安全，2017(5)：14-21.
PENG Jianshan, XI Qi, WANG Qingxian. Automatic exploitation of integer overflow vulnerabilities in binary programs[J]. Netinfo Security, 2017(5): 14-21. (in Chinese)

[9] DENG F L, WANG J, ZHANG B, et al. A pattern-based software testing framework for exploitability evaluation of metadata corruption vulnerabilities[J]. Scientific Programming, 2020, 2020: 1-21.

[10] ZHAO Z X, WANG Y, GONG X R. HAEPG: an automatic multi-hop exploitation generation framework [C]//Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. [S. l.: s. n.], 2020: 89-109.

[11] CHEN K, ZHANG C, YIN T, et al. VScape: assessing and escaping virtual call protections[C]//Proceedings of the 30th USENIX Security Symposium. [S. l.: s. n.], 2021: 1719-1736.

[12] 王瑞鹏，张旻，黄晖，等. 基于符号执行的格式化字符串漏洞自动验证方法研究[J]. 空军工程大学学报(自然科学版)，2021，22(3)：82-88.
WANG Ruipeng, ZHANG Min, HUANG Hui, et al. Research on automatic exploit generation method of format string vulnerability based on symbolic execution[J]. Journal of Air Force Engineering University (Natural Science Edition), 2021, 22(3): 82-88. (in Chinese)

[13] 刘凯. 二进制程序漏洞利用方法研究[D]. 青岛：中国海洋大学，2020.
LIU Kai. Research on exploitation method for binary program vulnerabilities[D]. Qingdao: Ocean University of China, 2020. (in Chinese)

[14] AAFER Y. Notes on non-executable stack[EB/OL]. (1986-09-02) [2023-04-14]. https://web.ecs.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf.

[15] BUROW N, ZHANG X P, PAYER M. SoK: shining light on shadow stacks[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy. [S. l.]: IEEE, 2019: 985-999.

[16] 魏强，韦韬，王嘉捷. 软件漏洞利用缓解及其对抗技术演化[J]. 清华大学学报(自然科学版)，2011，51

(10): 1274-1280.
WEI Qiang, WEI Tao, WANG Jiajie. Evolution of exploitation and exploit mitigation [J]. Journal of Tsinghua University(Science and Technology), 2011, 51(10): 1274-1280. (in Chinese)
[17] ASLR [EB/OL]. (2021-10-01)[2023-04-14]. http://pax.grsecurity.net/docs/aslr.txt.
[18] Position independent executable (PIE) performance [EB/OL]. (1994-05-26) [2023-04-23]. https://www.redhat.com/zh/blog/position-independent-executable-pie-performance.
[19] SHACHAM H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86) [C]//Proceedings of the 14th ACM conference on Computer and Communications Security. [S. l.: s. n.], 2007: 552-561.
[20] BELLARD F. QEMU, a fast and portable dynamic translator[C]//Proceedings of the Annual Conference on USENIX Annual Technical Conference. [S. l.: s. n.], 2005: 41-46.
[21] SCHWARTZ E J, AVGERINOS T, BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) [C]//Proceedings of 2010 IEEE Symposium on Security and Privacy. [S. l.]: IEEE, 2010: 317-331.
[22] 俞甲子, 石凡, 潘爱民. 程序员的自我修养:链接、装载与库[M]. 北京: 电子工业出版社, 2009.
YU Jiazi, SHI Fan, PAN Aimin. Self-cultivation of programmes, linker, loader and libraries [M]. Beijing: Publishing House of Electronics Industry, 2009. (in Chinese).
[23] LibFuzzer [EB/OL]. (2004-03-13) [2023-04-14]. http://llvm.org/docs/LibFuzzer.html.
[24] GAN S T, ZHANG C, QIN X J, et al. CollAFL: path sensitive fuzzing[C]//Proceedings of 2018 IEEE Symposium on Security and Privacy. [S. l.]: IEEE, 2018: 679-696.
[25] VISHNYAKOV A V, NURMUKHAMETOV A R. Survey of methods for automated code-reuse exploit generation[J]. Programming and Computing Software, 2021, 47(4): 271-297.
[26] 金继伟, 马菲菲, 张健. SMT 求解技术简述[J]. 计算机科学与探索, 2015, 9(7): 769-780.
JIN Jiwei, MA Feifei, ZAHNG Jian. Brief introduction to SMT solving[J]. Journal of Frontiers of Computer Science and Technology, 2015, 9(7): 769-780. (in Chinese)
[27] WANG F, SHOSHITAISHVILI Y. Angr—the next generation of binary analysis[C]//Proceedings of 2017 IEEE Cybersecurity Development (SecDev). [S. l.]: IEEE, 2017: 8-9.
[28] AFL [EB/OL]. (2021-01-18) [2023-04-14]. https://lcamtuf.coredump.cx/afl/.
[29] CHIPOUNOV V, KUZNETSOV V, CANDEA G, et al. The S2E platform: design, implementation, and applications[J]. ACM Transactions on Computer Systems, 2012, 30(1): 1-49.
[30] Pwntools[EB/OL]. (2014-08-13)[2023-04-14]. https://github.com/Gallopsled/pwntools.
[31] Z3[EB/OL]. (2007-10-09)[2023-04-14]. https://github.com/Z3Prover/z3.
[32] SEAN H, TOM M, DANIEL K. Automatic heap layout manipulation for exploitation[C]//Proceedings of the 27th USENIX Security Symposium. [S. l.: s. n.], 2018: 763-779.
[33] WANG Y, ZHANG C, ZHAO Z X, et al. MAZE: towards automated heap feng shui[C]//Proceedings of the 30th USENIX Security Symposium. [S. l.: s. n.], 2021: 1647-1664.

## 作 者 简 介

**黄 晖**
男,1987 年生,博士,讲师,研究方向为程序分析、网络安全
E-mail:huanghui17@nudt.edu.cn

**陆余良**
男,1964 年生,教授,博士研究生导师,研究方向为软件与系统安全、网络态势感知、大数据分析
E-mail:luyuliang@nudt.edu.cn

**朱凯龙**
男,1991 年生,博士,讲师,研究方向为网络安全
E-mail:zhukailong@nudt.edu.cn

**赵 军**
男,1976 年生,副教授,研究方向为网络安全
E-mail:zhaojun17@nudt.edu.cn

**责任编辑** 董 莉