|  |  |  |
| --- | --- | --- |
| 文档状态 | 保密级别 |  |
| [ √ ]草稿  [ ]修订  [ ]发布 | 文档编号 |  |
| 管理部门 |  |
| 修订年月 |  |
| 版本号 |  |

**BinAFL**

**可行性分析**

|  |  |  |
| --- | --- | --- |
| **修订人签字** | **审核人签字** | **批准人签字** |
| 日期： | 日期： | 日期： |

变更履历

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **序号** | **变更日期** | **版本** | **变更位置** | **变更原因** | **修订人** | **审核人** | **批准人** |
| 1 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

说明：“变更原因”主要是分为：

1. 建立初稿
2. 内容修订
3. 正式发布

目录

[1. 文档介绍 4](#_Toc201868817)

[1.1. 编写目的 4](#_Toc201868818)

[1.2. 文档范围 4](#_Toc201868819)

[1.3. 读者对象 4](#_Toc201868820)

[1.4. 术语与缩写解释 4](#_Toc201868821)

[1.5. 参考资料 4](#_Toc201868822)

[2. 项目介绍 4](#_Toc201868823)

[2.1. 项目说明 4](#_Toc201868824)

[2.2. 项目背景 5](#_Toc201868825)

[2.2.1．winafl的痛点 5](#_Toc201868826)

[2.2.2．winnie的痛点 5](#_Toc201868827)

[2.2.3．winfuzz的痛点 6](#_Toc201868828)

[2.2.4．Win消息通讯 6](#_Toc201868829)

[2.3. 项目目标 7](#_Toc201868830)

[2.4. 关键技术挑战 8](#_Toc201868831)

[2.4.1. 实现用户级快照需要攻克的技术难题 8](#_Toc201868832)

[2.4.2消息驱动模式执行效率的关键优化点 9](#_Toc201868833)

# 文档介绍

## 编写目的

快速浏览该项目的背景和技术难点。

## 文档范围

对前人工作的总结和现在工作的展望计划。

## 读者对象

Null

## 术语与缩写解释

| **术语或缩写** | **解释** |
| --- | --- |
|  |  |
|  |  |

## 参考资料

| **序号** | **文档名称** | **文档编号** | **版本** | **发布日期** |
| --- | --- | --- | --- | --- |
| 1 |  |  |  |  |
| 2 |  |  |  |  |

# 项目介绍

## 项目说明

本项目旨在构建一个革命性的、适用于Windows x64商业GUI软件的通用模糊测试平台。通过集成AI Agent技术与传统Fuzzing引擎，我们期望解决当前闭源软件Fuzzing中过度依赖人工逆向、无法模拟真实用户交互路径、以及存在大量无效崩溃（假阳性）的核心痛点。

## 项目背景

现在的Fuzzing都过于聚焦开源，对于闭源软件的Fuzzing研究甚少。当前比较主流的工作有winafl，winnie，winfuzz等，但他们的成功太过于依赖逆向出来的harness，而且他们专注的重点是单个函数的Fuzzing，就是跑出了crash，也很有可能在真实情况中，这个函数并不存在这样的参数输入，在大型程序中，他们会有一系列检查保证该函数的参数在一个合理范围内。

### 2.2.1．winafl的痛点

Winafl有两种模式，第一种模式就很简单，每次执行一个种子就启动一次程序。第二种模式是持久模式，并不需要每次执行一次种子都需要启动，它有一个函数循环，它会在目标函数返回时又回到进入到这个目标函数里面，这样会比第一种模式快很多，但是会产生大量的状态污染。

Program

Harness

Reverse

Target Func

Start

Loop

### 2.2.2．winnie的痛点

Winnie沿用了winafl的框架，它的主要贡献在于提供了一个根据污点数据交汇自动生成harness的解决方案，以及解决winafl的持久模式产生大量状态污染的问题。但是winnie的harness自动生成具有局限性，它不能跨dll分析，对于复杂的程序仍然需要手工的去构建harness，并且它fork的实现依赖于windows的固定版本，移植程度差。

Program

Harness Gen

Taint analysis

Harness

Sink common func

Target Func

Target Func

Fork

Exec finished

### 2.2.3．winfuzz的痛点

Winfuzz主要沿用了winnie的框架代码，它的主要贡献在于解决了winnie的fork移植性差的问题，它手动实现了程序快照。它的主要痛点还在于harness需要手动去构建。并且它的程序快照只支持简单的程序，对于复杂的GUI程序，这个快照的恢复是不可取的。

Program

Harness

reuse Winnie

Checkpoint

Record curr\_state

Target Func

enter

restore

### 2.2.4．Win消息通讯

任何一个Windows图形界面程序本质上都是一个被动的、由事件驱动的实体。程序的执行流程并非由自身代码逻辑从头到尾决定，而是依赖于一个持续运行的消息循环（Message Loop）来接收并响应来自操作系统、用户或其他程序的通知，这些通知就被称为消息（Message）。

其基本工作流程如下：

1. 事件产生: 用户进行操作（如点击鼠标、敲击键盘）或系统状态改变（如窗口需要重绘）。
2. 消息入队: 操作系统将这些事件转换为标准格式的消息（如WM\_LBUTTONDOWN、WM\_KEYDOWN、WM\_PAINT），并将它们投递到对应应用程序的消息队列（Message Queue）中。
3. 消息循环: 应用程序内部的主线程通过一个while循环，不断地调用GetMessage()函数从队列中取出消息，并通过DispatchMessage()函数将消息分派出去。
4. 消息处理: DispatchMessage()最终会将消息发送给操作系统，由操作系统调用目标窗口注册的特定处理函数，这个函数就是大名鼎鼎的WndProc。

WndProc (窗口过程函数) 是整个Windows消息机制的基石和最终目的地。它是一个遵循特定签名的回调函数，每个窗口在创建时都必须指定其对应的WndProc。当消息被分派到某个窗口时，实际上就是该窗口的WndProc函数被调用。

其原型通常为：LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

* uMsg: 核心参数，一个无符号整数，代表具体的消息类型，例如WM\_CREATE (窗口创建), WM\_COMMAND (用户点击按钮、菜单项), WM\_DESTROY (窗口销毁)。
* wParam 和 lParam: 消息的附加参数。例如，对于WM\_COMMAND，wParam通常包含了被点击控件的ID。

在原生的Win32编程中，WndProc通常表现为一个巨大的switch-case结构，根据不同的uMsg值执行不同的代码块。它的重要性在于，它是所有GUI交互的必经之路和逻辑中枢。任何对UI的操作，最终都会物化为一个或多个消息，并由WndProc处理。

MFC (Microsoft Foundation Classes) 作为一个主流的C++开发框架，它并没有改变消息驱动的本质，而是对其进行了优雅的面向对象封装。MFC通过其精巧的消息映射（Message Map机制，将庞大的switch-case结构分解到各个独立的成员函数中。开发者不再需要手动编写WndProc，而是通过BEGIN\_MESSAGE\_MAP等宏，将一个消息（如WM\_LBUTTONDOWN）同一个类的成员函数（如OnLButtonDown()）绑定起来。虽然形式上更加高级，但其底层依然依赖WndProc和消息分派。

传统GUI Fuzzing工具的痛点在于，它们或者只能进行“盲目”的像素点击，或者需要深入逆向才能找到关键的函数入口。而Windows的消息机制，为我们的AI Agent提供了一个具有高度可读性和结构化的中间层。其核心优势在于：消息类型是人类可读且具有明确语义的。一个WM\_COMMAND消息，其uMsg值虽然是一个数字（0x0111），但它在Windows SDK中被定义为具有特定含义的常量。我们的AI Agent，特别是基于大型语言模型（LLM）的Agent，可以理解这种映射关系。这样我们可以通过AI抽取出用户的操作消息，然后基于操作消息AI可以模拟用户行为操作。

## 项目目标

我们希望能通过AI和Fuzzing结合的技术，做出一个windows GUI程序的通用Fuzz，这个fuzz具有普适性，不需要大量的逆向工作，而且产生的crash具有真实性，而不会出现大量假阳。并且Fuzz的速度需要接近WINFUZZ的速度。

在我们设计的“AI Agent + Fuzzer”协同架构中，Windows消息是连接AI意图与程序行为的基础桥梁，然而，它在整个Fuzzing循环中扮演的具体角色，并非一成不变，而是由一个关键技术前提所决定的：我们是否能实现对目标GUI程序进行高效的用户级程序快照（User-level Program Snapshot）。这里的快照并非传统的虚拟机（VM）快照，而是指能够在进程内部瞬间冻结并保存其完整的执行上下文，包括所有线程、内存、句柄等状态，并能随时恢复，这类似于Linux系统中的fork()，但在Windows上对第三方闭源程序实现通用fork是极其困难的。

根据是否具备此能力，我们的Fuzzing平台将演化出两种截然不同的工作模式。在理想情况下，如果具备这种进程内快照能力，消息的核心角色将是深度分析与目标定位工具。AI Agent会通过发送消息或UI自动化来导航程序至高价值攻击界面，例如一个复杂的文件导入对话框，然后在触发核心功能的瞬间执行快照。此后，GUI和Agent的任务便已完成，Fuzzing引擎将在一个不涉及UI的极紧密循环中，反复从快照恢复状态、在内存中直接植入变异数据并恢复执行。因此，在此模式下，消息的重要性体现在Fuzzing循环的“前夜”，它帮助AI进行语义分析，自动化地找到创建快照的最佳时机，从而将缓慢的GUI导航与超高速的内存Fuzzing彻底分离，使效率逼近WINFUZZ。

然而，在更为现实的、不具备通用进程内快照能力的情况下，我们只能依赖较慢的VM快照或进程重启来重置状态，此时消息的角色便演变为路径导航器与攻击启动器的双重身份。在每一次Fuzzing迭代中，AI Agent都必须发送一整套完整的消息序列来重现用户路径，以确保程序能到达正确的攻击状态；紧接着，它再发送消息来注入Fuzzing数据，并发出最后一条“启动”消息来触发对恶意数据的处理。这就意味着，消息的重要性贯穿了每一次迭代的始终，它既是“铺路”的导航员，也是“扣动扳机”的执行者。虽然这种模式因GUI交互和状态重置的巨大开销而远慢于WINFUZZ，但其优点在于技术实现上更为直接和通用，更能保证项目的普适性目标。

下图，利用msg分析核心计算函数

Program

AI Analyzer

Intel PIN

Hijacker

Checkpoint

Target Func

RET

Usr operation

Data

Validate

Corr

Err

Snapshot

## 关键技术挑战

### 实现用户级快照需要攻克的技术难题

如果我们追求最快的fuzzing速度，我们需要解决在Windows闭源环境下进行进程状态完整重建的一系列难题。

1. 堆内存与句柄恢复 (Heap & Handle Restoration): 这是最核心的挑战。当我们在某个时间点创建快照时，目标程序的堆内存是一个包含了无数相互关联的内存块的复杂结构。简单地拷贝内存映像是不足够的，因为堆管理器（如ntdll!RtlHeap）自身的内部状态也必须被精确保存和恢复。而且在GUI程序中因为需要渲染，所以可能无时无刻都在进行大量的堆分配和堆释放，这样会映入噪声堆的问题。更棘手的是句柄（Handle），如文件句柄、窗口句柄（HWND）、GDI对象（画刷、字体）等。这些句柄本质上是内核空间的索引，一个单纯的用户态内存快照无法保存内核的完整状态。恢复快照时，如果这些句柄在内核中已失效，程序将立刻崩溃，这要求我们的快照方案必须有能力处理或“清洗”这些内核依赖。

2. 线程上下文与同步恢复 (Thread Context & Synchronization Restoration): 一个GUI程序通常包含多个线程。快照不仅要保存每个线程的CPU寄存器状态，还必须处理它们的同步问题。如果快照发生在某个线程持有锁（如CRITICAL\_SECTION）的时刻，在没有精确恢复内核同步对象状态的情况下强行恢复执行，极有可能导致整个程序死锁。因此，一个健壮的快照/恢复机制必须能识别并处理这些临界区状态。

3. GUI状态与资源一致性 (GUI State & Resource Consistency): Windows的GUI子系统（user32.dll, gdi32.dll）自身管理着大量的会话级（Session-level）资源。如果在快照与恢复的循环中，对这些资源（如设备上下文DC、GDI对象）的引用计数处理不当，会造成资源泄露。当泄露累积到一定程度，将导致整个Windows桌面管理器崩溃，而非仅仅是目标程序崩溃，这会严重影响Fuzzing的稳定性。

### 2.4.2消息驱动模式执行效率的关键优化点

如果我们规避了状态恢复的难题，但直面了另一个核心矛盾：如何在一个需要完整重现用户路径的模式下，最大限度地提升执行速度。 这要求我们必须对目标程序的执行流进行“外科手术式”的优化。

1． UI渲染剥离 (Detaching the UI Rendering): GUI程序最大的开销之一就是渲染，即将数据绘制到屏幕上。为了加速，我们必须让程序“假装”在运行UI，但跳过实际的渲染工作。这需要我们构建一个强大的API Hooking框架，对GDI/GDI+（如BitBlt, DrawText）、DirectX以及各种第三方UI库的渲染API进行拦截。当目标程序调用这些API时，我们的Hook函数可以直接返回成功，而不执行任何GPU或CPU密集型的绘制操作。这里的挑战在于Hook的全面性（覆盖足够多的渲染API）和兼容性（错误的Hook可能导致程序逻辑出错或崩溃）。

2. I/O及延迟操作优化 (Optimizing I/O & Delays): 另一个显著的瓶颈是磁盘I/O、网络I/O以及各种形式的“等待”。我们需要对相关API进行Hook和“欺骗”：

* 文件/网络I/O: 通过Hook ReadFile, WriteFile, send, recv等函数，可以将文件或网络操作重定向到内存缓冲区，或者直接返回一个模拟的数据，从而避免物理设备的延迟。
* 延迟函数: 必须Hook Sleep, WaitForSingleObject等所有可能导致程序挂起的函数，修改其行为，使其立即返回，从而“快进”掉所有不必要的等待时间。
* 挑战: 这种优化同样存在风险。某些程序的逻辑强依赖于Sleep带来的时序或I/O操作的真实结果。过于激进的优化可能会使程序偏离其正常的执行路径，从而错过只在特定时序下才会触发的漏洞。

## 设计方案

整个项目框架会有三大块组成：Agent，client，server。

我们首先会启动server端，server端最开始处在一个分析状态模式，它会记录记录和筛选用户的消息操作，以及记录程序当前状态，比如有多少个线程，多少个窗口，当前CPU/IO使用率，后把这些消息序列丢给agent去分析，agent会提取出来有用的消息需求，然后反馈给client端，client端会重放这些消息序列，观测程序状态，然后把程序状态反馈给agent，agent然后进一步得到更好的消息序列，并且初始化阶段也会有人类的参与，人类也会简单的参与消息的分析，比如agent会提取出不同的消息序列段，然后server端会把他们分类，然后依次播放，播放完之后还会询问人类是否保留这组消息序列。一旦筛选出这些消息序列，我们要么用agent对这些消息序列做变异，要么仅仅重放这些消息序列做完程序功能的启动。

一旦我们分析出相关的消息序列，我们的server端就会启动intel pin去分析这个消息序列激活的target function的函数入口地址，然后会把这个函数入口地址交给agent的ida pro服务器去分析这个函数的功能，如果这个函数agent认为满足target func，我们就会把这个函数加入hijack模块的备选队列中。然后hijack模块会劫持这个函数，会在这个函数的返回地址前加上一个状态清楚的功能函数。然后服务器会尝试启动一轮，如果程序出现崩溃或者异常，就认为没有找到这个target func的函数，就会清除之前所劫持的函数，然后会从队列中再劫持下一个函数。如果最后都没有找到target func，我们就进入侧信道判断这个功能操作的结束时刻。比如当CPU和IO低于一个阈值，我们就认为这个功能已经执行完毕了，就会清除相关状态，然后重新发送消息。

如果成功劫持目标函数之后，这时我们会进入两种fuzz模式的选择，一种fuzz模式是每次都会利用消息去触发这个target函数，一种模式是patch这个target func，让它变成winafl的持久模式，反复的循环。这取决于我们的状态清除函数写的有多好。

在劫持模块中，我们还会劫持各种IO相关操作的API。这个的目的不仅是为了加快fuzz的速度，也是为了变异做准备。首先，如果消息触发的功能需要外界的数据的话，它肯定会调用IO操作相关的API，我们以及劫持了这个API了，我们可以根据我们自己的需要去变异程序读到的外界数据文件的内容，每次我们读完之后会分配一片内存保存相关的文件数据，这样下次读取变异的时候，就不需要真的再从磁盘里读了，我们可以直接从内存中读取，加快程序的运行速度。

劫持模块还会劫持与一些无关紧要的API，这个可能需要agent模块去分析哪些API对于我们要测试的功能是无关紧要的，如果我们的专注的重点是程序的文件解包功能，我们可能就会nop一些网络或者渲染的API，然这些API并不真正进行计算，而是返回一个成功的空值，这样也会加快程序的运行速度。这种API的hook具有很高的特异性，需要人工和AI的参与。

在afl模块中，我们会沿用afl的一些随机变异策略，但是我们也并不是盲目的。Agent会根据相关IO API的调用情况，以及对相关数据的读取次数和读取顺序来决定我们需要变异的数据的优先级。我们会根据不同的优先级进行不同的变异策略。例如先读的文件数据很有可能是程序对这个文件的检查，如果我们想fuzz这个程序对于文件的边界检查，我们就会把这段数据地址的变异优先级提高。又比如高频的数据读取，我们可能认为这是程序这个功能的核心数据它的优先级也会大于其它普通数据。

State Recorder

State Restore

State Sanitizer

Agent

Msg Recorder

Msg Sanitizer

Hijacker

PIN

Analysis/Mutate

AFL Strategy

### 2.5.1．msg启动的设计方案

下图，利用msg启动程序功能

Program

AI Analyzer

Intel PIN

Hijacker

Target Func

Restore

RET

Replayer

Usr operation

Msg

Data

Validate

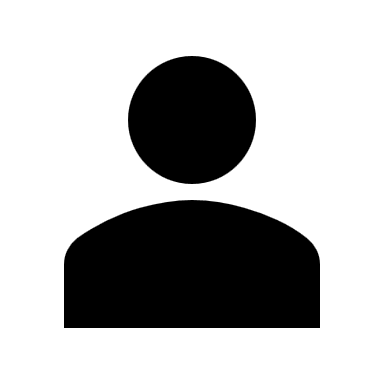
Corr

Err

### 2.5.2. Replayer模块设计

Replayer模块是用来重放AI提取出来的消息的。首先我们的服务端通过控制客户端记录用户的所有的消息，这些消息是会做噪音筛选的，如果相同的消息超过了10条，那么我们不会再记录这条消息了。然后10秒钟中，把这些消息永久到文件中，然后这个文件会在20秒后喂给AI，然后会提取出这个文件里面重要的msg，当AI完成提取msg的操作之后，我们会让AI发一条done的消息到BinAFLServer中，这个健壮性还是不够强，或许可以考虑在write之后，直接发送done的消息，不用AI去发是不是好一些？然后我们会尝试重放msg，每次重放msg之后（没有做hwnd是否还存在的检查），都会问询一下用户，如果这个消息成功激活了用户想要的功能，那么就会被加入到消息库中，作为重放消息的依据。

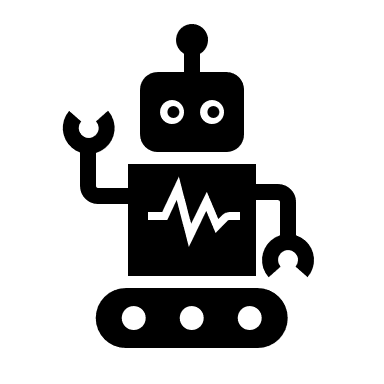
Program



Operate

Msg

noise filtering



CoreMsg

Usr Confirm

TrueMsg

Reply