一个组件安全自动化测试平台的设计与实现*)

陈锦富　卢炎生　谢晓东　游　亮　温贤鑫
(华中科技大学计算机科学与技术学院　武汉 430074)

摘　要　研制自动化的组件安全测试工具将对基于组件的软件工程产生重大影响，并能有效保障组件软件的安全性和可靠性，是当今软件业界一个极具现实意义和挑战的课题。针对广泛使用的微软第三方 COM 组件，设计和实现了一个组件安全性测试的原型系统 CSTS(Component Security Testing System)。CSTS 主要对组件从静态和动态两个级别进行安全性测试。在静态级别上，先分析出组件接口信息，然后对接口方法从参数个数、参数顺序、参数范围和参数类型等方面进行错误注入测试；在动态级别上，先执行测试驱动，然后对组件所依赖的内存、磁盘文件系统和注册表等环境进行错误注入，再通过强大的监测机制监视错误注入后组件执行情况来判断组件安全异常。所实现的原型系统 CSTS 具有自动化程度高、操作简单及测试效果较好等优点。

关键词　测试工具，COM 组件，安全测试，错误注入，动态监测

Design and Implementation of an Automatic Testing Platform for Component Security

CHEN Jin-fu　LU Yan-sheng　XIE Xiao-dong　YOU Liang　WEN Xian-xin
(School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract　The automatic testing tools of component security bring great effect on component-based software engineering, and they can effectively ensure the security and reliability of component-based software. Aiming at the third-party Microsoft COM component used widespread, a prototype tool called CSTS(Component Security Testing System) was designed and implemented. It can employ two levels' testing for component security, namely static testing and dynamic testing. In the static testing level, CSTS firstly analyzes component type information such component objects, methods and parameters etc. , and injects faults into interface method through parameter numbers, parameter sequence, parameter scope and parameter type for testing component. In the dynamic testing level, the tested component was firstly droved, and then the environment faults including memory fault, file system fault and registration fault etc. were injected into the tested component. Dynamic monitoring mechanism can monitor the component's runtime information after injecting the faults. CSTS can analyze the component exceptions by monitoring log file, and has high automation, good maneuverability and better testing ability.

Keywords　Testing tool, COM component, Security testing, Fault injection, Dynamic monitoring

1　引言

目前基于组件的软件工程(CBSE)已成为软件工程领域的研究热点。CBSE 是从面向对象的开发方法发展而来，其特点是能够实现组件的重用，缩短软件开发周期[1]。基于组件的软件开发技术进一步提高了软件开发效率及软件性能，但组件的可靠性和安全性问题始终没有得到较好的解决，并一直困扰着组件的开发方和使用方。诸多的组件软件开发组织在保障软件正确性、可靠性和安全性过程中仍是通过测试人员手工作业的方式，显然不能适应大规模软件测试的需要。设计完整有效的组件安全测试工具框架结构、研制自动化的组件安全测试平台是当今软件业界一个极具现实意义和挑战的课题。

软件测试的研究应从“理论＋技术＋辅助工具＋管理”4 个侧面展开，且它们之间互相依存，缺一不可。除了探索测试基本理论、发明新型测试技术、优化测试过程管理外，研制高效、便捷和可视化程度高的辅助测试工具(平台)尤为重要[2]。微软 COM 组件是当今 Windows 平台下最为广泛使用的组件之一。本文针对第三方 COM 组件，在基于我们提出的错误注入测试方法[3]和动态监测测试方法[4]的基础上，讨论如何设计、实现一个组件安全测试的原型系统。所实现的工具 CSTS 主要对组件从静态和动态两个级别进行安全性测试。在静态级别上，先分析出组件接口信息，然后对方法参数从参数个数、参数顺序、参数范围和参数类型等方面进行错误注入，注入错误测试用例后再编译执行，观察组件运行状态；在动态级别上，先执行测试驱动，然后对组件所依赖的环境进行错误注入，主要从内存、磁盘文件系统、注册表、进程及网络五个方面进行错误注入，再次执行并观察其监测信息。最后用 CSTS 工具对 38 个有漏洞的组件进行了测试分析，结果表明 CSTS 工具有较好的测试效果和性能。

*)国家预研项目(No. 513150601)。**陈锦富**　博士研究生，主要研究方向为软件测试、数据库系统；**卢炎生**　教授，博士生导师，主要研究方向为软件工程、数据库系统、数据挖掘；**谢晓东**　博士，讲师，主要研究方向为软件测试、数据库系统；**游　亮**　博士生，主要研究方向为软件测试；**温贤鑫**　硕士生，主要研究方向为软件测试。

2 系统总体框架

CSTS是在 WindowsXP 平台上用 Visual Studio. NET 2005 C#开发的一个针对 COM 组件安全性测试的集成测试系统原型，主要包含以下主要功能：(1) 对 COM 组件进行接口分析，得到必要的类型信息；(2) 自动生成组件方法级测试程序，通过进一步生成接口方法测试用例实现接口参数错误注入测试；(3) 编译测试程序，生成组件测试驱动；(4)对被驱动运行中的 COM 组件进行环境错误注入测试；(5)动态监测被测试 COM 组件的运行情况，记录组件安全异常，并写入日志文件；(6) COM 组件安全评估与分析。图 1 是 CSTS 系统的总体结构图。下面结合该结构图对主要的功能模块(步骤)作总体性的介绍，其关键实现技术在第 3,4 和 5 节中详细讨论。

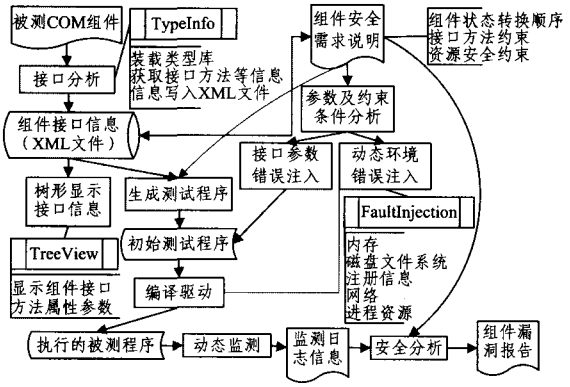


图 1 CSTS 系统总体结构图

2.1 接口分析模块

对于第三方 COM 组件，其源代码不可知，测试前必须通过一定的手段得到组件的类型信息。CSTS 系统中通过分析组件类型信息(TypeInfo)得到组件的接口方法等详细信息。类型库(TypeLibrary)包含 COM 组件所暴露接口的二进制描述信息。它们包含的信息与 IDL(接口定义语言)文件的基本相同。类型库可以是单独的二进制文件(. tlb)，也可以是动态链接库或可执行文件(. dll,. olb,. exe)中的资源。通过分析类型库可以获得以下信息：

(1)类型库名称和版本信息；

(2)类型信息个数；

(3)每个类型信息的接口类型(如 CoClass，Interface，Dispatch 等)、函数个数及变量个数；

(4)函数名、返回值类型、标志位、调用类型(如 function，get，put 等)、参数个数、可选参数个数、参数名、参数类型和参数方向。

以上分析得到的信息均同时存储在预先定义的 XML 格式文件中。

2.2 生成测试程序

根据接口分析模块产生的 XML 文件，生成测试用例模块所需要的测试程序(CS 测试文件)。测试文件的格式以 Simple2 组件为例，如图 2 所示。所有标记 TestFixture 的为一个测试用例集，标识 Test 为需要测试的接口方法，方法中为自动生成的测试用例。

主要实现步骤是：(1)使用 XPath 技术读取接口信息 XML 文件；(2)根据 XML 文件，利用 CodeDom 技术生成测试程序文件；(3)根据组件安全需求说明得到组件接口方法中的初始参数。

```
using System;
usfng System.Collections.Generic;
using System.Text;
namespace simple2test
{
  [TestFixture]
  public class Fun
  {
    simple2Lib.FunClass myFunClass = new simple2Lib.FunClass();
    [SetUp]
    public void Setup()
    {
    }

    [TearDown]
    public void TearDown()
    {
    }

    [Test]
    public void IFunAdd()
    {
      //测试用例
      long result = myFunClass.Add(1, 2);
      Assertion.Assert(result == 3, "equal");
    }
    [Test]
    public void IFunotherAddother()
    {
      //测试用例
      long result = myFunClass.Addother(2, 3);
      Assertion.Assert(result == 5, "equal");
    }
    [Test]
    public void IFunotherMultother()
    {
      long result = myFunClass.Multother(3, 4);
      Assertion.Assert(result == 12, "equal");
    }
  }
  [TestFixture]
  ...
}
```

图 2 测试程序文件

2.3 编译驱动测试组件

自动编译模块主要完成把待测试 COM 组件、测试用例文件和相关程序集(dll 格式)自动编译成动态链接库文件(dll 格式)，供驱动程序调用。实现功能：通过读取配置文件 config. xml 获取指定位置的 COM 组件和 COM TLB 文件，完成 COM 组件注册；选择测试用例程序(程序类型为 cs 格式)及相关程序集(dll 格式)，编译生成可执行的动态链接库文件，用于驱动器调用执行。

2.4 环境错误注入

对运行时的组件，从内存、磁盘系统、进程、网络及注册信息等相关环境进行错误注入。在组件被驱动运行时错误注入模块根据错误注入的类型在进程中对组件和操作系统交互的相应 API 进行拦截。采用的技术是用预先定义的伪 DLL 替换系统 DLL。根据环境错误注入测试要求，预先编写一些同名的伪 DLL 用于替换原来的系统 DLL。根据错误注入类型，对于不需要拦截的 API 函数直接把参数传递给系统 DLL 对应的 API 函数处理并返回结果。对需要拦截的 API，则在伪 DLL 中添加拦截代码进行拦截，同时植入错误。

2.5 动态监测

动态监测模块主要利用 Windows 操作系统中调试 API 技术获得对测试驱动的控制，从而对其运行状态进行监控。实现的主要功能有：(1)监测组件函数运行时的堆栈值(函数的输入参数)；(2)在函数运行结束时的堆栈值(函数的输出参数)；(3)CPU 寄存器的值(函数返回值)。基本原理：在获取组件函数的入口地址后，在函数入口上设置断点；当程序调用了设置断点的函数时便会产生调试中断事件，进一步可暂停目标程序的运行，读取目标程序内存，最终获得目标程序内存中希望得到的信息。通过分析内存信息及组件异常日志，可发现组件可能存在的安全漏洞。

2.6 安全评估与分析

安全分析的主要功能是根据组件安全需求说明及动态监测日志信息对组件进行安全分析，产生测试报告。存在潜在

安全漏洞分两种情况：(1)异常日志中明确标引的接口方法及异常代码；(2)没有被触发异常但组件内部运行状态变化违反了安全需求说明中的约束条件。

3 组件接口信息的分析、存取及显示

通过获取组件或类型库的绝对路径可装载类型库，进而可在不创建组件对象的情况下分析并提取组件的接口信息。其步骤如下：

Step 1 获取被测试组件的绝对路径。

Step 2 装载类型库，并得到组件接口的 ITypeInfo 接口指针。装载类型库通过函数 LoadTypeLib()实现，并通过 GetTypeinfoGuid 函数获得组件对象的 ITypeInfo 接口指针。

Step 3 接口方法等信息的提取。

接口分析可以获得 4 个层次的信息：组件对象、接口、方法、参数。分析时每一层次对下一层进行循环遍历访问提取。使用到的主要方法有：

LoadTypeLib()：获得 ITypeLib 指针；

ITypeLib::GetTypeInfo()：获得 ITypeInfo 指针；

ITypeInfo::GetTypeAttr()：获得 TypeAttr 结构体指针；

ITypeInfo::GetRefTypeOfImplType()，ITypeInfo::GetRefTypeInfo()：获得组件对象实现的接口信息；

ITypeInfo::GetFuncDesc()：获得 FUNCDESC 结构体指针；

ITypeInfo::GetVarDesc()：获得 VARDESC 结构体指针。

分析到的接口信息存储到预先定义的 XML 格式文件中，XML 文件的格式如图 3 所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Components SYSTEM "ComponentTypeInfo.dtd">
<Components>
  +<ComponentConfiguration></ComponentConfiguration>
   <ComponentObjectNumbers></ComponentObjectNumbers>
   <ComponentObjects>
    <ComponentObject>
      <ComObjectName></ComObjectName>
      <ComponnetInterface>
          <InterfaceNumbers></InterfaceNumbers>
              <Interfaces>
                  <Interface>
                      <InterfaceName>InterfaceName</InterfaceName>
                      <MethodNumbers></MethodNumbers>
                       <Variables></Variables>
                        <Methods>
                              <Method>
                                 <MethodName>MethodName</MethodName>
                                 <MethodParams></MethodParams>
                                 <OptionalParams></OptionalParams>
                                 <ReturnType>ReturnType</ReturnType>
                                 <InvokeKind>Function</InvokeKind>
                                 + <Parameters>
                                 </Parameters>
                              </Method>
                        </Methods>
                  </Interface>
              </Interfaces>
      </ComponnetInterface>
    </ComponentObject>
   </ComponentObjects>
  +<SecurityRequirementSpecification>
   </SecurityRequirementSpecification>
</Components>
```

图 3 存储接口信息 XML 格式

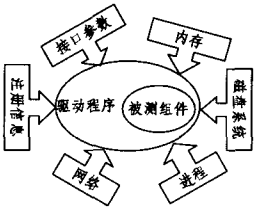
存储 XML 文件使用的技术主要有：Dot NET 为 SAX 提供的 XmlWriter 类技术(只向前、未缓存的流模型)和 Dot NET 中 DOM 的实现方式 XmlDocument 类技术(将整个 XML 文档在内存中以树形表示)。存储时先用 XmlWriter 类写流格式的 XML 文件模板，再用 XmlDocument 类编辑 XML 文件：迭代查找、插入和修改 XML 节点。

采用方法 XML2TreeView(TreeView)将 XML 文件中的类型信息转换为树形视图并显示在窗体中。主要技术是迭代 XML 节点，将其加入 TreeView 中。

4 错误注入测试技术

错误注入(Fault Injection)技术作为一种测试技术是指按照特定的故障模型，用人为的、有意识的方式产生故障，并施加于待测系统中，以加速该系统的错误和失效的发生[5,6]，同时观测并反馈系统对所注入故障的响应信息，通过分析，对系统进行验证和评价的过程。1998 年普度大学 WenLiang Du 和 Aditya P. Mathur 的技术报告“Vulnerability Testing of Software System Using Fault Injection”认为“系统”由“应用程序”和“运行环境”组成[7]。所有被认为不属于运行程序的代码就属于环境。相应的系统错误注入方法可分为应用程序错误注入和环境错误注入。环境错误注入是一种自动化的软件测试方法，这种方法在协议安全测试等领域中都已经得到了广泛的应用[8-10]。一个完整的错误注入系统应该尽可能多地对漏洞错误进行全面测试，对其尝试注入各种可能的错误，从而达到测试出软件缺陷的目的[11]。

应用程序用户除了人作为可见用户外，还有一些不可见用户，如 API、操作系统、文件系统等[12]。根据这个观点，在 CSTS 测试系统中采用图 4 所示错误注入模型。错误注入模型 FIM[3]是一个六元组{IP, M, DF, PRS, NET, REG}，其中 IP 为接口参数、M 为内存、DF 为磁盘文件系统、PRS 为外部组件或进程、NET 为网络、REG 为组件注册信息。规定了对待测试组件进行错误注入的 6 个方面。实际应用中根据不同待测试对象的不同测试需求，FIM 可以进行扩充。根据 COM 组件的特点，CSTS 系统采用静态接口参数错误注入和动态环境错误注入相结合实施错误注入测试，多方面测试 COM 组件的安全性。接口分析可以得到组件的对象、接口、方法及方法参数等必要的错误注入测试信息，然后对组件方法进行错误注入。环境错误注入是指在组件运行时对组件和操作系统交互环境进行错误注入。其错误注入层次如图 5 所示。



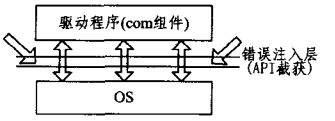


图 4 组件安全错误注入模型　　图 5 环境错误注入框架

接口参数处的错误注入首先分析组件接口信息，然后对输入参数、参数类型、参数个数、参数顺序等进行错误注入。内存错误注入主要是限制组件访问的内存大小、低内存注入、内存访问控制、内存出错等内存相关的模拟错误注入。磁盘文件系统错误注入主要是限制对文件的读写访问、文件损坏、目录损坏、磁盘写保护等文件系统相关的模拟错误注入。进程错误注入主要有进程资源不足、调用组件不能找到、dll 文件损坏等进程资源错误注入。网络错误注入主要是在网络线路及网络组件上进行模拟错误注入，如网络断线、拒绝服务、服务延时、网络拥塞等相关错误注入。注册信息错误注入主要有恶意修改注册表信息、拒绝修改、键值损坏、注册表加密等相关错误注入。

5 动态监测技术

CSTS 系统中采用了基于组件异常的动态监测方法。对

于 Windows 组件而言，通过捕获函数 DebugActiveProcess 连接到当前组件所在的测试驱动进程。任何时候被调试的进程发生异常都会产生 EXCEPTION_DEBUG_EVENT 事件[4]。可能的异常包括试图访问不可访问的内存，执行断点指令，试图被 0 除，或者任何其它 SEH 处理的异常等[13]。系统能监测到的主要异常范围如表 1 所示。这些异常都会被捕捉到，从而能够及时感知组件出现错误并进一步跟踪错误。发生异常时需要对现场数据进行保护，现场记录的信息包括：线程上下文、程序加载的模块信息、调用栈信息、虚拟栈信息。将这些信息进行保存，可以进一步对错误定位。

表 1　监测的主要异常范围

类型	异常项及原因
内存相关异常	存取越界：运行线程试图对一个内存地址进行读或写，却没有相应的存取权限
	数组访问越界：线程试图存取一个越界的数组元素
	存取不存在的内存页：由于文件系统返回一个读错误，造成不能满足要求的页故障
	访问被保护的页面：一个线程试图存取一个带有 PAGE_GUARD 保护属性的内存页
	线程的栈空间溢出：线程用完了分配给它的所有栈空间
	执行非法指令：线程执行一个在当前机器模式中不允许指令或线程执行一个无效的指令
算术相关异常	除数为 0：线程试图用 0 来除一个整数或浮点数
	操作数范围越界：一个运算操作的结果超过了数值规定的范围
	浮点栈溢出：由于浮点操作造成栈溢出或下溢
内存泄漏	通过监视 malloc，free，realloc 函数的输入输出值，并进行统计，可实现内存泄漏检查

利用 Windows 调试 API 进行监测的主要步骤如下：

(1)使测试驱动进程进入被调试状态

对于一个已经启动的测试驱动进程，利用 DebugActiveProcess 函数就可以捕获测试驱动进程，使测试驱动进程进入被调试状态。DebugActiveProcess 函数原型是：BOOL DebugActiveProcess(DWORD dwProcessId)，参数 dwProcessId 是测试驱动进程的进程 ID。若要启动一个新的测试驱动进程，通过 CreateProcess 函数设置必要的参数便可使测试驱动进程进入被调试状态。主要通过设置函数 CreateProcess 的 dwCreationFlags 属性为 DEBUG_PROCESS 或 DEBUG_ONLY_THIS_PROCESS。

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union
  {
    EXCEPTION_DEBUG_INFO Exception;
    CREATE_THREAD_DEBUG_INFO CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO ExitThread;
    EXIT_PROCESS_DEBUG_INFO ExitProcess;
    LOAD_DLL_DEBUG_INFO LoadDll;
    UNLOAD_DLL_DEBUG_INFO UnloadDll;
    OUTPUT_DEBUG_STRING_INFO DebugString;
    RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

图 6　调试消息数据结构

(2)对测试驱动进程进行监视

测试驱动进程进入了被调试状态，调试程序(监视程序)负责对被调试的程序进行调试操作的控制。调试程序通过 WaitForDebugEvent 函数获得来自测试驱动进程的调试消息。调试程序根据得到的调试消息进行处理，测试驱动进程将暂停操作，直到调试程序通过 ContinueDebugEvent 函数通知测试驱动进程继续运行。函数 ContinueDebugEvent 参数 lpDebugEvent 中可以获得调试消息。图 6 显示了调试消息数据结构，其中 dwDebugEventCode 记录了产生调试中断的消息代码。图 7 显示了主要的消息代码。

```
EXCEPTION_DEBUG_EVENT: 产生调试例外;
CRATE_THREAD_DEBUG_EVENT: 新的线程产生;
CREATE_PROCESS_DEBUG_EVENT: 新的进程产生;
EXIT_THREAD_DEBUG_EVENT: 一个线程运行中止;
EXIT_PROCESS_DEBUG_EVENT: 一个进程中止;
LOAD_DLL_DEBUG_EVENT: 一个 DLL 模块被载入;
UNLOAD DLL DEBUG EVENT: 一个 DLL 模块被卸载。
```

图 7　主要的调试消息代码

在得到测试驱动进程的调试消息后，调试程序根据这些消息代码进行不同的处理，最后通过函数 ContinueDebugEvent 通知测试驱动进程继续运行。

(3)对测试驱动进程设置断点

监测的目标是组件接口函数的输入输出及函数运行时组件的内存状态等信息。因此对组件接口函数的地址设置断点成为整个监测的关键。通过分析 COM 组件的类型库可以获得 COM 接口函数的入口地址，进一步可设置断点。设置断点分两种类型：1)在函数入口地址处设置断点。将 0xCC(int 3)写入指定的内存地址便可设置断点。当测试驱动进程运行到指定地址时，将产生调试中断信息并通知调试程序。这里用到读写指定进程内存的 API 函数 WriteProcessMemory，ReadProcessMemory。此外，由于操作系统对进程空间的程序代码段进行了保护，所以要使用函数 VirtualProtectEx 修改页码保护属性。2)设置函数返回地址的断点。在函数入口断点中断时可以取得函数返回值地址，进一步可以设置返回地址的中断。

(4)处理调试中断

测试驱动进程运行到断点地址(组件函数入口与返回处)时将产生断点中断，此时测试驱动进程暂停。通过 GetThreadContext(&context)得到进程中断处的线程上下文。此时 context. esp 就是函数的返回地址，context. esp 所指位置值就是函数运行时的栈顶。栈中保存了函数运行时的参数，通过 ReadProcessMemory 函数就能得到运行时的参数值。同理，通过返回时断点处的线程上下文信息可以得到函数运行时的返回值(保存在 context. eip 中)。在每次断点发生时可以获取进程运行的具体函数，并通过读取进程的内存空间值得到组件的运行状态。此外，根据需要可以设置测试驱动进程单步运行，从而可以跟踪进程的每一步操作。

整个监测过程中监测机制负责监测组件运行状态并动态收集相关状态信息。开始时日志信息为空，随着组件的运行，它保存了一系列状态变量、内存变化和资源使用情况等安全相关信息[4]。具体保存的信息有 TimeStamp，ComponentID，PID，MethodName，ConstrainCondition，Function，Type，ReturnVal，Param，Memory，Handle，Bytes Read/Writeen 及 ExceptionInformation 等。测试系统 CSTS 根据状态监测日志中的组件状态信息及其相关异常信息识别组件是否存在安全漏洞。

6　系统实现及集成测试环境

6.1　系统实现

系统实现时，接口分析把分析得到的信息存储在预先定

义好的 XML 格式文件中，同时以树状显示在集成界面上。错误注入实现中 FIM={IP,M,DF,PRS,NET,REG}，对于每个错误注入因素扩展取若干个参数(错误注入项)。其中，IP={超出整型数范围，超出字符串长度，整型换成字符型，字符型换成整型，实参多于形参个数，实参少于形参个数}，M={段锁住，内存不足，无效地址，无法访问，页文件太小，内存大小限制}，DF={磁盘已满，文件只读，文件已存在，文件未发现，文件已锁住，文件数据错误，文件拒绝访问，文件正在使用，文件无法创建}，PRS={未发现进程文件，库存文件不充足，无效文件类型，访问拒绝，COM 组件不能注册，系统 DLL 不能加载，进程损坏}，NET={网络断开，无可用端口，网络 API 未发现，网络延时，网络拥塞，拒绝服务，连接无效}，REG={访问拒绝，注册表损坏，键值不存在，注册表 IO 失败，键无效，值无效，注册表不能写}。动态监测机制主要采用了基于 Windows Debug API 监测技术。动态监测机制能动态记录被测组件运行情况及组件异常信息，从而利用异常分析机制可进一步定位异常触发的位置及原因。

6.2 集成测试环境

CSTS 可以测试含有 TLB 信息或附带独立 TLB 文件的 COM 组件。针对每个测试对象，CSTS 均会建立一个以 ctw 文件管理的项目，用于保存测试过程中的中间结果、临时文件和配置信息。根据用户的需要，可以对同一测试组件展开不同的测试工作，在 CSTS 中表现为不同的测试操作。测试操作可以看作是一个测试执行过程，即编写测试用例、错误注入、动态监测、产生测试日志等。集成测试环境中一个完整的测试过程如图 8 所示。

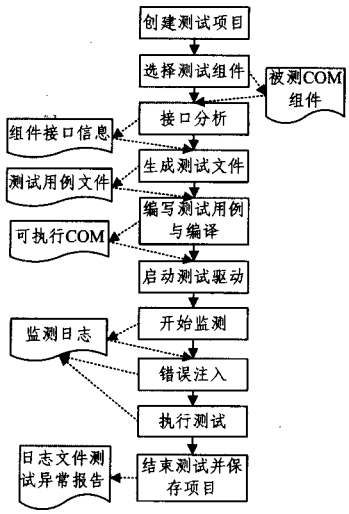


图 8 一个完整的测试过程

CSTS 在接口分析得到 XML 格式存储的接口信息基础上，解析出组件对象、组件接口、组件方法以及方法参数等信息。将组件接口的层次信息以树状直观显示在集成测试环境中，供测试用户编写方法测试用例。在编写测试用例与编译过程中，用户可以只对关心的某一个或几个方法编写测试用例，而不理会其它方法，增加了测试的灵活性。编写测试用例完成后可直接编译，编译信息将详细地显示在测试环境消息框中，依据编译信息可完成编写符合语法的测试用例。CSTS 可以直接驱动编译成功的测试组件，驱动后便可进行动态监测和错误注入。当一次错误注入完成后可执行测试，根据测试结果和监测信息可多次错误注入并执行测试，直到达到用户预期效果为止。当结束测试驱动后可对异常信息进行分析并评估组件安全情况，测试日志可同时保存以备日后分析和查询。图 9 是 CSTS 正常操作模式下的集成环境快照。

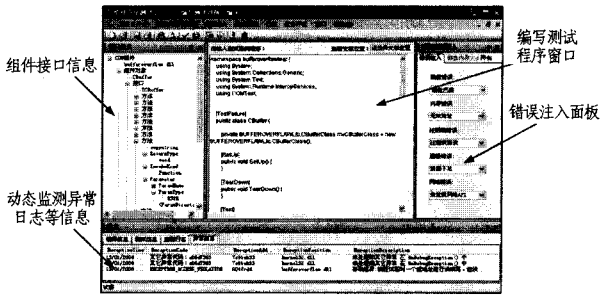


图 9 自动化测试工具 CSTS 快照

7 测试结果分析

CSTS 通过对每次执行(编写测试用例、生成测试驱动、环境错误注入)进行动态监测并记录测试行为，最终对这些监测记录进行归纳、分析，然后生成组件异常信息测试报告。异常信息报告包括异常发生时间、异常代码、异常地址、异常发生所在函数及异常描述等统计条目。

为了测试 CSTS 系统的效果和性能，我们对 CSTS 进行了反复实验。在 CVE(Common Vulnerabilities and Exposures)等一些安全网站上收集了 38 个有漏洞的组件进行测试。先对被测试组件进行针对性的错误注入，然后用动态监测机制记录监测日志及异常信息。选择错误注入项测试组件时，可选择 6 种错误类型中的任一种，也可以组合选取多种错误注入项进行测试。一般来说，多种错误注入项比单一错误注入项更容易触发组件异常。实验结果如表 2 所示。其中，3 因素组合错误注入触发了最多的组件异常，检测率达到 97.37%；两因素组合错误注入触发的组件异常数少于 3 因素触发的异常数，检测率为 73.7%；单因素错误注入触发的异常数最少，检测率为 39.47%。进一步分析可知，导致组件异常的各错误注入因素中，接口参数错误注入占 71.6%。大部分组件异常均由于注入超过方法参数范围的输入参数值导致缓冲区溢出，从而触发组件异常。此外，内存错误也是触发组件异常的一个重要因素，占 15.2%，仅次于接口参数错误。如果由手工分析并触发组件异常，则将花费大量的时间和人力编写测试用例和记录测试过程，且检测率远远不如自动化工具的检测率。测试过程及效果将无法接受。实验结果表明 CSTS 系统具有较好的漏洞检测率。

表 2 CSTS 系统测试结果

错误注入因素	被测组件数	出现异常组件数	检测率
1 因素注入	38	15	39.47%
2 因素注入	38	28	73.7%
3 因素注入	38	37	97.37%

结束语 在基于组件的软件开发与维护活动中，测试第三方组件的可靠性和安全性是必不可少的一环，也是其中最为耗时、容易忽视的一个环节。研制自动化或半自动化的组件安全测试工具、提高组件安全测试效率是当今基于组件的软件工程中的一个亟待解决的课题。以目前广为使用的微软 COM 组件作为测试对象，提出了一个能进行接口分析、自动

(下转第 261 页)

可以看到线程 threadID:801189 并不是一开始就形成环路,而是受到后面那些线程的影响。我们输出整个占有-请求链而非只输出死锁环路部分,是为了更好地跟踪线程的行为。

结束语 在信息采集部分,只涉及 Hash 以及链表的插入、删除操作,这个过程的时间复杂度是 $O(l)$。而在查找环路部分,因为要对结构进行检索(4.2 节中步骤③),这个过程的时间复杂度是 $O(n^2)$。所幸系统中某时刻的用户线程数总是受限的,而结构中这些线程的平均链表长度也不可能过长,所以算法执行时间不会太长,得到的结果也比较令人满意。我们说过,此算法只在调试状态下执行。如果想要依托 4.1 节的结构对正常运行中的程序进行实时监控,结构的检索算法部分还要进行改进,以期在更少的时间片内完成一次检索,才有可能得到真实可信的结果。还有一个遗留问题是对 Event 对象的处理,可以尝试的一个思路是依据 Event 构件对象本身的自描述性质,在程序的语义分析阶段解决这一问题,那将是一个有趣的课题。

参 考 文 献

[1] Koretide. Elastos 2.0 Operating System Manual[EB/OL]. 2007. http://www.koretide.com.cn

[2] Koretide. CAR's Manual[EB/OL]. 2007. http://www.koretide.com.cn

[3] Tanenbaum A S, Woodhull A S. Operating Systems Design and Implementation Third Edition[M]. Beijing: Publishing House of Electronics Industry, 2006: 48-60

[4] Python Documentation, 15.3.3 Condition Objects[EB/OL]. 2006. http://docs.python.org/lib/condition-objects.html

[5] Richter J. Programming Applications for Microsoft Windows Fourth Edition[M]. Beijing: China Machine Press, 2005

[6] Microsoft MSDN Library, .NET Development 类库, System.Threading[EB/OL]. 2007. http://msdn2.microsoft.com/zh-cn/library/system.threading.aspx

[7] Coffman E G, Elphick M J, Shoshani A. System Deadlocks. Computing Surveys, 1971, 3: 67-78

[8] Levine G N. Defining Deadlocks. Operating Systems Review, 2003, 37: 54-64

[9] Bacon J, Harris T. Operating Systems Concurrent and Distributed Software Design[M]. Beijing: Publishing House of Electronics Industry, 2003: 297-316

(上接第 233 页)

编译、错误注入和动态监测的自动化工具的总体架构。所实现的测试工具原型 CSTS 主要具备如下优点:

(1)强大的环境错误注入功能。从内存、文件、注册表、进程和网络 5 个方面对组件依赖的环境进行模拟错误注入,最大可能地触发组件安全异常。

(2)全面的动态监测功能。对组件运行中的组件加载、方法调用、参数、返回值、返回地址、异常信息及异常地址等信息进行全面的监测和记录,特别是对异常信息,CSTS 将对其进行单独显示和分析。

(3)工具的自动化程度高,仅需要少量的人工参与。

(4) 测试工程以项目形式管理,有利于组件测试的组织和管理。通过测试项目的有效管理能达到部分测试资源(如用例)的重用,同时可再现历史测试活动,有利于测试结果的对比和分析。

(5) 测试环境可视化程度高。提供了直观的接口信息树状图显示、测试用例编写、方便的错误注入面板和监测信息列表等界面。测试人员在集成测试环境下根据提示便可轻易完成整个测试流程。

当前,我们也只是在组件安全测试自动化方面作了一些有益的探索,所实现的原型工具尚有一些不足之处:

(1) 对一些没有提供或集成 TLB 类型库的 COM 组件无法分析得到接口信息。

(2) 对一些如指针的指针、用户自定义类型等复杂方法参数类型,工具无法自动生成初始的测试用例代码,需要测试人员手工编写测试用例程序。

(3) 工具没有提供完整的组件安全等级评估机制。

参 考 文 献

[1] 毛澄映,卢炎生. 构件软件测试技术研究进展. 计算机研究与发展,2006, 43:1375-1382

[2] 毛澄映,卢炎生,谢晓东,等. 一个 C/C++程序集成测试平台的设计与实现. 小型微型计算机系统,2007,28:1037-1043

[3] Chen Jin-fu, Lu Yan-sheng, Xie Xiao-dong. Testing Approach of Component Security Based on Fault Injection // 2007 International Conference on Computational Intelligence and Security (CIS'2007). IEEE Computer Society, Harbin, China, 2007: 763-767

[4] Chen Jin-fu, Lu Yan-sheng, Xie Xiao-dong, et al. Testing Approach of Component Security Based on Dynamic Monitoring // Second International Multi-Symposiums on Computer and Computational Sciences(IMSCCS 2007). IEEE Computer Society, Iowa City, IA, USA, 2007: 381-386

[5] Hsueh M-C, Tsai T K, Lyer R K. Fault Injection Techniques and Tools. IEEE Compute, 1997, 30: 75-82

[6] Jeffrey V. Fault Injection for the Masses. IEEE Computer, 1997, 30: 129-130

[7] Du Wenliang, Mathur A P. Vulnerability Testing of Software System Using Fault Injection. Coast TR 98-02. 1998: 1-20

[8] Voas J, Mcgraw G. Software Fault Injection: Inoculating Programs Against Errors. John Wiley and Sons, 1997

[9] Looker N, Munro M, Xu J. A Comparison of Network Level Fault Injection with Code Insertion // the 29th IEEE International Computer Software and Applications Conference. Scotland, 2005: 479-484

[10] Whittaker J A, Mottay F E, Ei-Far L K. Testing Exception and Error Cases Using Runtime Fault Injection. Florida Tech Computer Science Technical Report. 2001

[11] Thompson H H, Whittaker J A, Mottay F E. Software security vulnerability testing in hostile environments // the 2002 ACM Symposium on Applied Computing. Madrid, Spain, 2002: 260-264

[12] Whittaker James A. Software's Invisible Users. IEEE software, 2001, 18(6): 84-88

[13] Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks // the Proceedings of the 7th USENIX Security Symposium. San Antonio, Texas, 1998: 63-78