doi:10.3969/j.issn.1000-436x.2014.09.003

# 基于异常控制流识别的漏洞利用攻击检测方法

王明华[1,2]，应凌云[1]，冯登国[1]
（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，北京 100049）

**摘 要**：为应对 APT 等漏洞利用攻击的问题，提出了一种基于异常控制流识别的漏洞利用攻击检测方法。该方法通过对目标程序的静态分析和动态执行监测，构建完整的安全执行轮廓，并限定控制流转移的合法目标，在函数调用、函数返回和跳转进行控制流转移时，检查目标地址的合法性，将异常控制流转移判定为漏洞攻击，并捕获完整的攻击步骤。实验结果表明，该方法能够准确检测到漏洞利用攻击，并具备良好的运行效率，可以作为漏洞利用攻击的实时检测方案。

**关键词**：软件漏洞；漏洞利用；攻击检测；地址随机化；数据执行保护
**中图分类号**：TP399 **文献标识码**：A **文章编号**：1000-436X(2014)09-0020-12

# Exploit detection based on illegal control flow transfers identification

WANG Ming-hua[1,2], YING Ling-yun[1], FENG Deng-guo[1]
(1. Laboratory of Trusted Computing and Information Assurance, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China;
2. University of Chinese Academy of Sciences, Beijing 100049, China)

**Abstract:** In order to deal with exploit attacks such as APT, an approach was proposed to detect exploits based on illegal control flow transfers identification. Both static and dynamic analysis methods were performed to construct the CFSO (control flow safety outline), which was used to restrict the targets of control flow transfers occurred during the target program's running. When a call/ret/jmp was about to execute, the target was checked according to the CFSO. The illegal control flow transfer is considered as an exploit attack and all the following attacking steps could be captured. The experiment also showed that proposed method had decent overhead and could be applied to detect exploits online.

**Key words:** software vulnerability; exploit; attack detection; address space layout randomization; data execution protection

.

## 1 引言

软件漏洞是信息系统安全的主要威胁，各大软件厂商在不断改进和完善软件开发质量管理，但软件漏洞问题仍无法彻底消除。根据 Secunia 发布的漏洞数据[1]，可以看到应用软件漏洞占绝大多数，并且漏洞数量逐年增多。此外，应用软件漏洞种类多样，包括 Flash 漏洞、浏览器漏洞、文件格式漏洞等，涉及到 Adobe Reader、Microsoft Office Word、Adobe Flash Player 等被广泛使用的软件。

为了缓解应用软件漏洞带来的危害，新型操作系统引入了地址空间随机化（ASLR）和数据执行保护（DEP）等安全机制，一定程度上抑制了针对应用软件和系统漏洞的利用攻击。但是，攻击者仍然能够通过精妙的利用构造，找到绕过这些安全机制的方法，通过劫持控制流实施利用攻击。为了应对这些新安全环境下的利用攻击，安全人员需要及时地发现攻击威胁，以便快速做出响应，避免造成损失。近年来，学术界提出了控制流完整性检测、污点分析等分析方法，来对程序执行过程中的异常

**收稿日期**：2014-07-17；**修回日期**：2014-08-30
**基金项目**：国家重点基础研究发展计划（“973”计划）基金资助项目（2012CB315804）；国家自然科学基金资助项目（91118006）；北京市自然科学基金资助项目（4122086）
**Foundation Items:** The National Basic Research Program of China (973 Program)(2012CB315804); The National Natural Science Foundation of China (91118006); The Natural Science Foundation of Beijing (4122086)

控制流进行检测，取得了一定的效果。控制流完整性检测方法[2]，通过在 CFG 图中构造控制流转移的合法目标地址集合，在控制流转移发生时，校验目标地址是否在合法的集合内，并以此作为攻击检测的依据。此种方法依赖 CFG，无法解决动态生成代码相关的恶意控制流转移问题，如 JIT Spraying 攻击等。动态污点分析方法[3~5]将程序输入标记为污点源，通过监控程序动态执行过程，在污点数据被异常使用时产生警告。由于污点分析需要细粒度地监控程序执行，使得系统运行效率开销很大，同时污点分析方法本身存在一些局限性，会使得检测结果存在一定范围的误报率和漏报率。在工业界，如 FireEye[6]等安全厂商提出了通过 API 行为拦截，利用一定的启发策略来判定恶意行为的方法。然而，这些检测方法在函数级别开展分析，但是漏洞利用攻击可能发生在指令级别，仅通过 API 拦截无法获知指令级的异常控制流转移，导致此类方法检测漏洞利用攻击的能力有限。

本文提出了一种针对应用软件漏洞利用攻击的检测方法：通过对目标程序加载的二进制模块进行静态分析，获得函数边界和导出属性等，构建初始的合法控制流转移边界；结合在程序动态执行时实时维护的控制流转移记录，构建完整的控制流转移安全轮廓。将程序执行过程中，转移至此轮廓之外的控制流转移，判定为具有利用攻击威胁的异常的控制流转移。本文方法通过识别异常控制流转移，在攻击代码执行之前，检测到利用攻击。本方法能够准确检测包括动态生成代码在内的恶意控制流转移的攻击；另外，本方法不依赖于任何漏洞和攻击代码的先验知识，同时具有较为理想的运行效率，可以作为漏洞攻击实时检测的解决方案。

本文方法的贡献与创新点如下。

1) 提出了一种通过检测异常控制流转移判定漏洞利用攻击的方法，结合静态和动态分析手段，构造完整的程序安全执行轮廓，识别安全轮廓之外的异常控制流转移，检测漏洞利用攻击。该方法较传统利用污点分析等细粒度数据流分析方法具有较高的准确性和较理想的运行效率。

2) 提出了一种检测来自于动态生成代码的利用攻击的方法，通过分析内存的页面属性、数据相似性等，对动态生成代码的合法性进行判断，从而识别转移至非法动态生成代码的控制流。该方法解决了控制流完整性校验等方法无法处理的诸如 JIT Spraying 攻击的检测问题。

3) 实现了漏洞利用攻击检测原型系统，并通过对 Word、IE、Flash Player 等 8 个实际漏洞进行检测实验，验证了本文方法的有效性。实验表明，本方法不仅具有很高的检测准确性，同时还具有较小的运行开销，可用于漏洞利用攻击实时检测。

## 2　相关工作

漏洞利用攻击检测相关研究主要包括基于动态污点传播的方法和基于控制流完整性的方法 2 大类。在动态污点传播方面，相关的漏洞攻击检测系统有 Argos[7,8]。Argos 将网络数据作为污点源，在动态执行过程中监控是否有 sink 函数使用污点。该系统能够用来检测 0 Day 漏洞攻击，能够支持针对 Windows 7 等操作系统上应用软件的漏洞攻击。该分析系统依靠污点分析实现，由于污点方法自身存在的局限性，所以难以保证监控结果的准确性，具有一定的误报和漏报率，同时具有较大的运行开销。相似的检测系统还有 TaintCheck[4]、Panorama[9]等。

基于控制流完整性校验（CFI，control flow integrity）的方法同样可以用于检测漏洞利用攻击。该方法[2]首先由 Abadi 等人提出，通过构造 CFG 图中所有间接控制流转移的合法目标地址的集合，在控制流转移发生时校验目标地址是否在合法的集合内，以此可以作为攻击检测的依据。然而，对于控制流转移结构复杂的程序而言，构造 CFG 的准确性难以保证，这会对检测结果有一定的影响。

binCFI[10]也是基于 CFI 方法实现，提出了间接控制流转移目标地址的完整性校验方法。这些目标地址包括地址指针常量，经过算术运算得出的程序地址，以及函数返回地址等。对这些目标地址的校验是基于静态分析的结果，会引入一定的误判，例如，方法没有验证函数返回的合法性，可能通过导致覆盖函数地址控制执行流的情况。这种异常情况需要结合动态执行信息来进行判断。

Total-CFI[11]同样利用 CFI 手段来判断间接控制流转移的合法性，是动态分析方法。该方法利用了影子栈，监测来自在用户态和内核态的控制流异常攻击，系统具备较理想的运行效率；但是，该系统在检测间接函数调用目标时粒度过粗，会带来一定的误判，同时对动态生成代码分析的较弱，无法检测通过 JIT Spraying 等实施的漏洞攻击。

CCFIR[12]通过重写二进制文件来加入控制流转移校验的过程，也具备漏洞检测的能力。CCFIR 在二进制文件中添加新的 springboard 节区，用以随机存放该模块中所有间接控制流转移目标地址，并改写间接控制流转移调用处代码，使得在这些代码执行时，首先跳转至 springboard 验证合法性。该方法能够抵御传统的 ret-to-libc 和 ROP 攻击，但是无法识别动态生成代码有关的控制流合法性。FPGate[13]提出的方法与 CCFIR 相似，但更关注函数调用的合法性。FPGate 和 CCFIR 等方法基于二进制重新技术（binary rewriting）来实现，需要用户重新部署新的二进制模块，适用性较差。

## 3　问题描述和方法框架

Windows 7 等操作系统采用了 ASLR 和 DEP 等安全机制，传统的函数地址覆盖、异常处理覆盖等漏洞利用手段已经难以奏效。要成功利用 Windows 7 等平台上应用软件的漏洞，首先需要突破 ASLR 和 DEP 等机制的防护，常见的方法包括 ROP[14]和内存喷射(spraying)[15]。ROP 能够通过重用内存中已加载模块中的代码片段，达到修改 shellcode 数据所在内存页面属性的目的，使数据执行保护失效；内存喷射技术通过在内存某段区域中大量部署 shellcode，从而在 ASLR 导致地址不确定的情况下，也能保证利用攻击具有较高的成功率。图 1 中给出了 2 种利用攻击过程。

通过这些攻击方式可以发现，漏洞得以成功利用是由于程序执行过程中发生了异常控制流转移。在图 1(a)中，shellcode 存放在 0x0c0c0c0c、0x7c345678、0x7c349654 等地址处的指令序列为 ROP 链中的 garget。攻击者首先利用 ROP 链，将 shellcode 所在内存区域处变为可执行，再将控制流转移至 shellcode，实施攻击。从图中可以看出，漏洞利用攻击所依赖的异常控制流转移发生在函数返回时，此时的控制流不应该转移至 0x7c345678 处的 ROP garget，而是应该转移至 call eax 调用时压入的下条指令地址处。在图 1(b)中，攻击者将 vtable 虚函数表中第 3 个虚函数被篡改为 0x0c0c0c0c，并且在内存中喷射出很多包含了攻击代码的内存区域，在函数调用 call[eax+0x8]时，目标地址不应是 0x0c0c0c0c 处的动态生成代码，而应该是由 ecx 所指定类中的虚函数。

根据上述分析，本文提出一种通过检测异常控制流转移判定漏洞利用攻击的方法。方法架构如图 2 所示。本方法通过对二进制目标程序进行静态分析，构造初始的安全执行轮廓；在动态监控程序运行阶段，识别动态生成代码，对安全执行轮廓进行维护。结合构建的完整安全执行轮廓，检测函数调用、函数返回、间接跳转等控制流转移的合法性，将异常控制流转移判定为漏洞利用攻击，进而捕获完整的攻击步骤。本文方法不但能够检测由于违反静态 CFG 图中限定的控制流转移所导致的利用攻

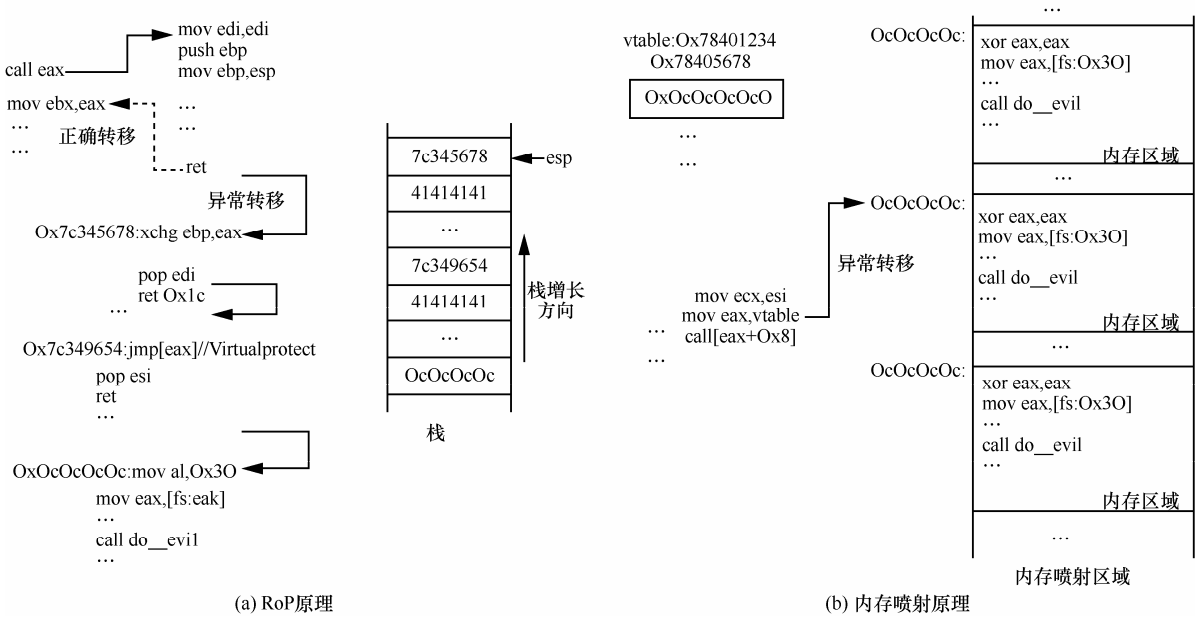


图 1　ROP 和内存喷射利用原理示意

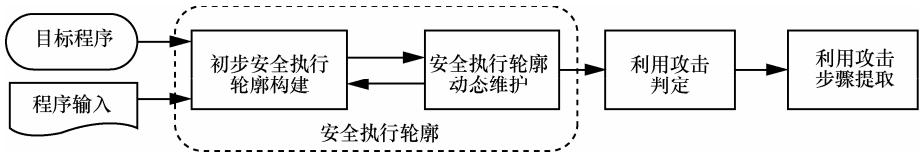


图 2　漏洞利用攻击检测方法架构

击，而且支持动态生成代码相关的利用攻击。

本文接下来将针对如何构建安全执行轮廓，如何判定漏洞利用攻击，以及如何提取漏洞利用攻击步骤进行详细阐述。

## 4　安全执行轮廓构建

安全的控制流转移轮廓，需要确保程序所运行的所有控制流转移指令能够转移至合法的目标地址。Abadi 等人提出的控制流完整性检验方法[2]，通过静态获得的 CFG 图，构造所有控制流转移的合法目标地址。然而，程序在运行过程中可能出现动态生成代码，例如：

1) 一些第三方软件通过加壳等方式进行保护，在软件运行时，首先开辟一块具有可执行属性的内存区域，然后将代码拷贝至区域中，程序首先执行此段动态生成的代码，然后进行后续的执行；

2) 某些程序在运行过程中，可能在函数起始地址等处添加 hook，在此函数执行时，程序将跳转至存放在另一个具有可执行权限的内存区域中，执行 hook 处理代码，如在 IE 8 进程启动并加载 IEShims.dll 模块时，将在一些模块函数入口处添加 hook，劫持这些函数的运行，以完成所需要的处理；

3) 另外，某些应用程序支持 JIT（just in time）代码运行，例如 Flash 程序，可以通过 Action Script 编写 Flash 程序，在程序运行时，Flash 代码被加载到可执行属性的内存页面中，通过 JIT 方式执行。

上述动态生成代码执行情况，Total-CFI[11]、binCFI[10]等基于 CFI 的方法均无法处理。为此，本文通过静态分析构造初始安全执行轮廓，在程序动态执行阶段，识别合法的动态生成代码区域，构建完整的控制流转移安全轮廓，从而不但能够判定模块之间的控制流转移的完整性，而且能够处理动态生成代码的安全控制流转移问题。

### 4.1　安全执行轮廓构定义

本文识别程序运行过程中所有可执行代码，将这些代码所在模块和函数作为安全执行轮廓，并基于此，判定程序运行时控制流转移的安全性。

为了阐述方便，做出如下定义。

1) 代码块 *tb*：每一个代码块由若干条顺序执行的指令组成，其中，仅最后一条为控制流转移指令，如 call、ret 和 jxx 等。它是进行控制流转移检测的最小检测单元。

2) 加载模块 *m*：是指待分析程序在导入表中引入的模块，或者在运行时动态加载的可执行模块；将加载模块构成的集合记为 $M_L$，即 $M_L = \{m\}$。

3) 模块函数 $f_m$：是指存在于模块 *m* 中的函数；将所有模块函数的集合记为 $F_L$，$F_L = \{f_m | m \in M_L\}$。

4) 动态生成代码（DGC，dynamic generated code）：在程序运行过程中，在具有可执行内存属性页面上、非加载模块中的代码；即 DGC=$\{tb | tb$ in $Pe$ && $tb$ not in $m$, $\forall m \in M_L$，$Pe$ 为具有可执行属性的页面$\}$。本文将 DGC 所在内存区域视为一个 DGC 模块 $m_{dgc}$，将 DGC 中可以被调用的一个执行单元记为 DGC 函数 $f_{dgc}$。具体 DGC 模块和函数的提取方法见 4.3 节。

5) 安全执行轮廓（CFSO，control flow safety outline）：具体包括模块轮廓和函数轮廓。其中，模块轮廓为 *M*，$M=M_L \cup \{m_{dgc}\}$；函数轮廓为 *F*，$F=F_L \cup \{f_{dgc}\}$。

安全执行轮廓的构建分为 2 个阶段：首先依赖二进制分析程序自身，构造初始的安全执行轮廓，此轮廓是静态 CFG 图中限定的所有执行函数 $F_L$ 和加载模块 $M_L$；然后，通过动态监控程序执行，识别所有 DGC 模块$\{m_{dgc}\}$和 DGC 函数$\{f_{dgc}\}$，对安全执行轮廓进行补充和维护。下面，首先讨论初始安全执行轮廓构建的过程，然后讨论如何通过动态监控目标程序执行，构建完整的安全执行轮廓。

### 4.2　初始安全执行轮廓建立

依靠二进制目标程序自身构造初始安全执行轮廓。主要包括获得二进制目标程序自身和其加载模块的属性，以及这些模块中所有函数信息。

对于函数，需要通过反汇编二进制模块获取每一个函数的起始地址。除此之外，还需要确定

模块中函数是否具有外部调用属性，即此函数是否允许被其他模块代码调用，例如模块 A 中代码调用模块 B 中函数，那么，此函数应该被模块 B 导出。为此，将函数分为 2 类，分别是外部调用函数和内部调用函数。具有外部调用属性的函数，可以被除自身模块之外的其他模块调用，例如模块导出的函数，C++类实例中具有 public 属性的虚函数等；具有内部调用属性的函数，则只能够被自身模块调用。

从模块的导出表和重定位表来获得具有外部调用属性的函数。通过导出表，可以获得此模块中所有导出的函数。通过重定位表，获得地址需要重定位的函数。具体方法是：依次扫描每一个重定位表项，如果该表项的数值与模块基地址的和在模块代码段的范围之内，那么就视其为一个地址需要重定位的函数。本文将这类函数和导出函数认为是具有外部调用属性的函数。最后，模块中除去这些外部调用属性函数之外的函数，认为是具有内部调用属性的函数。

通过分析函数外部调用属性，能够有效防范篡改虚函数地址的漏洞利用攻击。由于在虚函数表中的函数地址需要被重定位，所以，所有虚函数都被划归为外部调用函数。如果类实例的某个虚函数被覆盖为某恶意地址，并且此恶意地址不在此模块的外部调用函数集合之内，那么能够检测到这一异常控制流转移。

对于漏洞程序运行所依赖的模块，按照 PE 格式解析模块文件，获得模块加载基地址和大小，是否具备随机化属性，以及导入、导出、重定位表等数据项的内容。由于程序在运行时可以动态加载模块，对这些模块，虽然可以动态解析模块文件，获得上述模块和其中所有函数属性信息，但是这将使得系统运行效率受到较大的影响。所以，对系统中所有模块文件进行预先分析，并将分析结果保存到文件中。在后续动态分析判定过程中可以直接读取，从而提高了系统运行效率。

### 4.3 安全执行轮廓动态维护

为了处理 DGC 相关的控制流转移，需要动态识别 DGC 模块和 DGC 函数，并将其加入到安全执行轮廓中。虽然可以通过对二进制程序在运行时采取细粒度的跟踪分析，来获得 DGC 模块和 DGC 函数在内存位置，提取 DGC 数据，但是这需要较大的运行开销。相反，选择在 DGC 执行时，识别 DGC 模块和 DGC 函数。所用算法伪代码如下。

**算法 1** IdentifyDGC($M$, $F$,$M_L$, $F_L$,$tb$)

//该算法识别 DGC 模块和 DGC 函数，并加入到模块轮廓 $M$ 和函数轮廓 $F$ 中

输入：当前的模块轮廓 $M$ 和函数轮廓 $F$，进程加载模块集合 $M_L$，模块函数集合 $F_L$，当前执行的 $tb$

输出：最新的模块轮廓 $M$ 和函数轮廓 $F$

IF $tb$ not in $m$, $\forall m \in M_L$ : THEN

g_*bIsDGC* = true;

$m_{dgc}$ = GetMemRgn();

*eip* = &*tb*; //*eip* 是当前 *tb* 的首条指令地址

IF check_attacky(*eip*) == true: THEN alert;

ELSE $M=M \cup \{m_{dgc}\}$; $f_{dgc}=f_{dgc} \cup \{tb\}$;

ENDIF

ELSE

IF g_*bIsDGC*== true: THEN

$F = F \cup \{f_{dgc}\}$; g_*bIsDGC*=false;

ENDIF

ENDIF

该算法在每一个 *tb* 执行时被调用。函数参数 $M_L$、$F_L$ 是通过构建初始安全执行轮廓时得到的。算法检测当前执行代码是否在已加载模块内（第 1 行），若不在，则说明当前正在执行 DGC，接下来提取 DGC 模块和 DGC 函数（第 2 行之后）。

由于系统随机化不仅针对加载模块，而且包括例如 TEB、PEB 等关键数据结构、栈和堆分配地址的随机化，所以，攻击者在考虑到利用攻击的稳定性，会通过喷射的方式保证攻击的成功率。为此，充分利用这一特性，识别 DGC 模块，并区分其合法性和非法性。

本文将内存区域认为是具备相同属性的相邻内存页面的集合。那么，在利用攻击发生时，寄存器 eip 值一定在某个存放了 shellcode 等恶意代码的具有可执行属性的内存区域之内。这个区域相邻的内存区域，一定也存放了与此内存区域高度相似的恶意代码数据。这些内存区域是攻击者为了保证利用攻击的稳定性，大量喷射操作的结果。本文通过检测相邻内存区域中数据的相似度，来判断控制流转移至 DGC 的合法性。此检测过程由算法 1 中 check_attacky 实现。算法伪代码如下。

**算法 2** check_attacky(*eip*)

//该算法检测 *eip* 所在的 DGC 是否具有攻击特征

输入：当前程序执行的 pc

输出：如果具有攻击特征，输出 true；否则为 false

*RgnLst* = GetMemRegionList();

<*RgnFmr*, *RgnCur*, *RgnLtr*>=GetNeighborRgns (*eip*, *RgnLst*);

IF *RgnCur.exec==RgnFmr.exec &&*

*RgnLtr.exec=RgnCur.exec &&*

*RgnCur.exec==EXECUTABLE:* THEN

*off = eip – RgnCur.base;*

*insts_cur* = GetInstData(*eip*, *N*);

*insts_fmr* = GetInstData(*RgnFmr.base+off, N*);

*insts_ltr* = GetInstData(*RgnLtr.base+off, N*);

IF *d* = chk_similarity(*insts_cur*, *insts_fmr*, *insts_ltr*) > *T* : THEN

IF chk_shellcode_like(*insts_cur*): THEN

RETURE true;

ENDIF

ENDIF

ENDIF

RETURN false

算法首先获取进程内存空间中所有内存区域 *RgnList*，然后找到当前 *eip* 指向的区域 *RgnCur*，以及与之相邻的 2 个区域 *RgnFmr* 和 *RgnLtr*。如果这些区域不都具备可执行属性，那么当前内存区域不是喷射而来。否则，计算出要执行的指令地址在当前内存区域中的偏移，并且验证在相邻内存区域上相同偏移处的数据相似性。算法中，设置了一个检测窗口 *N*，用于表示在各个内存区域中读取的字节数量，并用变量 *d* 表示数据相似性。如果 *d* 高于某一个阈值 *T*，即认为高度相似，那么再反汇编当前内存区域的指令序列，验证指令序列中是否存在疑似 shellcode 的指令，最后返回验证结果。

由于 shellcode 的形式具有多样性，例如被多次加密或存在多个攻击指令片段等，使得验证 shellcode 疑似指令具有一定的误报率；另外，验证 shellcode 会对分析运行效率造成一定影响，所以，在相邻内存区域都具备可执行属性，并且相同偏移处数据高度相似之后，再验证 shellcode 疑似指令。这种策略的可靠性在于，攻击者为了提高漏洞利用的成功率，会通过内存喷射等手段在内存页面中部署尽可能多的攻击代码，使得相邻的内存区域中的数据具有高度的相似性。

由于 shellcode 需要重定位获得加载地址，以及可能需要实时获得执行所需 API 地址，所以通过检测重定位指令和一些访问进程数据结构的敏感指令序列，来判定当前指令数据疑似 shellcode 的程度。其中，重定位指令包括 call/pop、fnstenv/pop、fxsave/pop 等常见指令组合，敏感指令包括获得当前进程控制块、获得当前进程加载模块的指令序列等。

如果经过上述检验，此内存区域不具备攻击属性，那么将此内存区域作为合法的 DGC 模块，并获取此区域的基地址和大小作为 DGC 模块的基地址和大小，将此 DGC 模块加入到维护的可执行模块的集合中。在程序执行过程中，此段 DGC 可能重复执行，集合元素的唯一性保证了此 DGC 模块在可执行模块集合中仅出现一次。相反，如果检验发现此内存区域具有攻击属性，那么当前转移至此的 DGC 控制流非法，判定为漏洞利用攻击。

对于 DGC 函数，由于无法像模块函数那样获知 DGC 函数的起始地址，所以，当程序控制流由非 DGC 模块转移至某段 DGC 时，将此次控制流转移的目标地址视为 DGC 函数起始地址，而在控制流由 DGC 模块返回至非 DGC 模块时，视为此段 DGC 函数执行结束，并将在 DGC 模块之间执行的所有代码块视为一个 DGC 函数体。并且，将 DGC 函数视为具有外部调用属性的函数。DGC 函数示意如图 3 所示。

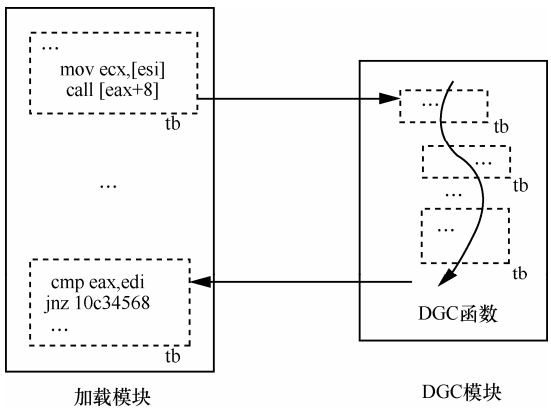


图 3　DGC 函数示意

## 5　漏洞利用攻击判定

根据构建的安全执行轮廓，对程序运行过程中发生的间接控制流转移进行验证，将不合法的控制流转移判定为漏洞利用攻击。为此，对 3 种间接控制流转移进行分析：通过 call 指令进行的函数调用；

通过 ret 执行进行函数返回；以及通过 jmp 执行进行程序跳转。下面分别就这 3 种控制流转移的验证方式进行讨论。

### 5.1 函数调用的检测方法

间接调用的目标地址仅能在运行时确定，需要根据构建的安全执行轮廓来验证目标地址的合法性。如果间接函数调用目标地址不在当前调用模块之内，首先验证目标地址是否是函数的起始地址，再验证此目标函数是否具有外部调用属性；如果目标函数被所在模块导出，则还需验证调用模块是否导入了此模块。如果间接函数调用目标在当前调用模块内，仅验证目标地址是否是当前模块的函数，不限定此函数的属性。只有通过验证的间接函数调用，才被认为是合法。上述验证过程的算法伪代码如下。

**算法 3** chk_call(*dst*, $m_c$)

//检测 call 指令目标地址的合法性

输入：call 指令的目标地址 *dst*，call 指令所在模块的代码段 $m_c$

IF *dst* in $m_c$: THEN

chk_is_func($m_c$, *dst*) ? pass : alert;

ELSE

<$f_x$, $m_x$> = get_function(*dst*)

IF chk_is_func($m_x$, $f_x$) && ($f_x$∈ $m_x$.*EAT* && $m_x$ in $m_c$.*IAT*)

‖ $f_x$.is_external:

THEN pass;

ELSE alert;

ENDIF

ENDIF

工作 Total-CFI[11]同样检测 call 控制流转移合法性的问题，但与本文方法不同的是，它仅仅验证 call 的目的地址是否是一个函数的起始，而不关心这个函数是否具有外部调用属性，这导致验证结果不够准确，可能会产生漏报。另外，此系统还无法处理 DGC 相关的控制流转移问题。

### 5.2 函数返回的的检测方法

控制流可以通过函数返回指令进行转移。正确的函数返回的目标地址，应该是此函数被调用时压入栈中的地址。在函数返回时，如果验证发现返回地址不合法，可以判定控制流被劫持，即有异常控制流转移。

与 Total-CFI 系统类似，通过影子栈来实现返回地址合法性的验证。为每一个线程维护一个影子栈，在函数调用发生时，在对应的影子栈中压入返回地址；在函数返回时，在对应影子栈中验证压入的地址与此返回地址是否一致。考虑到异常处理、setjump/longjmp 等特殊情况，返回地址可能不在栈顶，但是一定在栈中。所以，在本文中仅验证返回地址是否在影子栈中，并将栈顶到此项之间所有的项弹出影子栈。如果未在栈中，则判定为异常控制流转移。

另外，与 Total-CFI 不同的是，本文还考虑了另一种特殊情况：push addr、ret。这种 ret 实际上起到 call 的功能。由于 addr 值是由 push 压入，而非函数调用 call 压入，所以在影子栈中无法找到。本文通过额外为每一个线程维护一个 push 栈来记录压入的值，在函数返回 ret 发生时，首先查看该线程的影子栈，如果没有在栈中找到返回值，则在 push 栈中寻找。如果找到，将此值至栈顶的元素全部弹出。否则，则说明当前 ret 可疑，可能使程序控制流转移至危险位置。

### 5.3 跳转转移的检测方法

跳转可分为直接跳转和间接跳转。对于直接跳转，目标地址一定在自身模块内，无需检测。对于间接跳转，需要检测目标地址的合法性。

对于 jmp [*C*+*idx***idx'*]模式的跳转，其中，*idx*、*idx'*为寄存器或立即数，如果 *C* 值在程序代码段或者数据段范围内，那么认为此 *C* 值是程序中某个函数地址表的起始，其中的每个地址项占用 4 个字节，此跳转指令正是通过地址表 *C* 获得偏移 *idx***idx'*处的目标地址。本文将此类跳转指令合法目标地址集合记为 *S*

$S=\{C[i] \mid C[i] \text{ in } m_c \text{ \&\& } C[N+1] \text{ not in } m_c, i=0, 1, \cdots, N\}$

即从地址表 *C* 开始，依次查看每一表项的值是否在当前模块的代码段 $m_c$ 范围内，并将符合此条件的值加入到 *S* 中，直至遇到不满足此条件的地址项为止。在此类 jmp 执行时，如果目标地址不在其合法地址 *S* 中，将此指令的执行判定为异常控制流转移。

对于其他间接 jmp 跳转指令，目标地址理论上可以是内存中任意位置，但通过对实际应用程序的分析，发现合法的间接跳转指令的目标地址都应在本模块内。所以，将跳转至自身模块代码段之外的间接跳转，判定为异常的控制流转移。

## 6　漏洞利用攻击步骤捕获

在捕获到利用攻击所需的异常控制流转移之后，捕获后续漏洞利用攻击的步骤。这些攻击步骤有助于分析人员掌握攻击细节，对评估漏洞的危害性，以及制定防御方案等具有重要的意义。

当识别到异常控制流之后，开始单步执行目标程序，记录程序执行指令和调用的函数。对于执行的指令，记录每一条指令类型以及操作数的值；对于调用的函数、记录参数和返回值。通过监控进程创建、用户创建、文件操作、网络访问等系统 API 调用，能够捕获“下载并运行”、“添加新用户”等恶意行为。

通过这些记录，分析人员能够还原漏洞利用攻击的所有细节。例如，如果漏洞利用采用了 ROP 方式，可以获知为了构造 ROP 链，利用代码重用了哪些指令片段，攻击载荷所在内存区域如何获得可执行属性，以及攻击载荷所进行的恶意操作。再如，通过检测利用攻击点所在内存区域相邻的内存区域的属性，以及比对这些区域的数据相似性，可以获知内存喷射漏洞的喷射粒度和喷射范围等信息，评估此漏洞利用的成功率等。安全分析人员可以利用这些信息，对软件漏洞的危害性进行评估，制定漏洞补丁方案，或者生成检测工具所需的漏洞特征等。

## 7　实验评估

基于硬件模拟器 QEMU 1.6.1[17,18]实现原型系统 ECfield (enhanced control flow integrity based exploit detector)，并利用 8 个实际漏洞利用攻击样本对本文的系统进行实验评估。在实验中，原型系统运行在配备了四核 3.20 GHz Intel Core i5-3470 CPU、8 GB 内存、250 GB 硬盘的 Fedora Core 13 计算机上。实验漏洞攻击样本运行在 QEMU Guest 系统中。各个样本运行的 CVE 编号、漏洞程序和所运行的系统环境如表 1 所示。

### 7.1　系统实现

ECfield 包括静态分析和动态分析 2 个组件。在静态分析组件中，对二进制加载模块和模块内的函数进行分析，构建初始的安全执行轮廓；在动态分析组件中，根据执行状态，对安全执行轮廓进行维护，同时，对控制流转移进行验证，将不合法的控制流转移判定为利用攻击，并提取攻击步骤。

构建初始的安全执行轮廓时，通过在 IDA Pro 6.1[19]中编写 IDA Python 插件，提取二进制漏洞程序和其加载的动态链接库中的所有模块和函数属性等信息。模块信息包括加载地址、大小、导入表、导出表和重定位表、随机化属性等数据信息；函数属性信息包括所属模块，函数起始地址相对模块加载地址的偏移，函数的外部调用和内部调用属性等。本文将这些分析结果以文件形式保存，在动态分析阶段，系统通过读入文件内容到内存中，进行后续分析和验证操作。

在 QEMU 中添加进程识别、线程识别、模块识别等功能模块，分别获得系统中新创建的进程、每一个进程所创建的线程、以及每一个进程加载的模块。在系统运行过程中，根据当前代码块所在内存地址，判定是否为 DGC。如果属于 DGC，遍历进程虚拟空间描述符 VAD，获得相邻内存区域，判定这些区域的数据相似度。当数

**表 1**　　**实验样本**

| CVE 编号 | 漏洞软件 | 运行环境 |
|---|---|---|
| CVE-2007-4607 | Easymail 3.0.61 | 32 bit Windows 7 专业版 |
| CVE-2010-3333 | Microsoft Office Word 2003 | 32 bit Windows 7 专业版 |
| CVE-2010-2883 | Adobe Reader 9.3.4 | 32 bit Windows 7 专业版 |
| CVE-2011-0611 | Flash Player 10.0.42.34 | 32 bit Windows 7 专业版 |
| CVE-2012-0158 | Microsoft Office Word 2007 | 32 bit Windows 7 专业版 |
| CVE-2012-4969 | Internet Explorer 8 | 32 bit Windows 7 专业版 |
| CVE-2013-2551 | Internet Explorer 8 | 32 bit Windows 7 专业版+SP1 |
| CVE-2014-1761 | Microsoft Office Word 2010 | 32 bit Windows 7 专业版+SP1 |

据高度相似时，通过验证当前执行的 DGC 指令序列是否具有 shellcode 特征，来判定当前 DGC 是否具有攻击属性。如果具有攻击属性，则直接判定为利用攻击；否则，提取 DGC 模块和 DGC 函数，加入到安全执行轮廓中。在验证相邻内存区域数据相似度时，将检测窗口长度 $N$ 设置为 32 byte，并且相似度阈值 $T$ 设置为 80%。另外，从 Metasploit[20]中提取验证过程中所用到的 shellcode 疑似指令特征。

在系统运行过程中，为目标程序中每一个线程维护一个影子栈，验证所有发生在用户态的函数返回的合法性。当间接函数调用发生时，通过安全执行轮廓，验证目标地址是否是目标模块中具有外部调用属性的函数。当间接跳转时，根据跳转指令的模式，验证目标地址是否在合法的跳转地址集合内、或是否在当前模块代码段范围之内。如果验证不合法，则判定为控制流转移异常。

在检测到异常控制流转移后，ECfield Hook 进程创建、用户添加、网络访问等系统 API，并开启对目标程序的单步跟踪分析。通过改变 QEMU 代码块译码逻辑，使其每个代码块仅包含一条指令，实现单步执行的效果。在每个代码块执行时，ECfield 检测当前地址是否是被 Hook 函数的起始地址。如果是，就从栈中读取函数参数和返回地址。在被 Hook 的函数返回时，ECfield 读取 QEMU Guest 系统的 eax 寄存器获得函数返回值。单步执行的指令信息和函数调用信息，最终保存在文件中，以便供分析人员还原攻击细节。

### 7.2 利用攻击检测结果

本文选取了近年具有较严重威胁的漏洞，对实验系统进行评估。这些漏洞涉及用户广泛使用的软件，如 IE、Adobe Reader、Word、Flash Player 等，漏洞的利用攻击类型也涵盖多种，包括 ret-to-libc、SEH exploit、ROP 攻击、Heap Spraying、JIT Spraying 等。实验中检测到的各个样本程序的异常控制流转移指令地址、控制流转移指令、攻击类型等信息如表 2 所示。

从上述漏洞利用攻击中可以看到，CVE-2012-0158 这一针对 Word 2007 的利用攻击仍然是通过直接覆盖函数返回地址，采用 ret-to-libc 的方式实现的。此攻击能够成功的原因在于，即使 DEP 机制已经部署到 Windows 7 系统中，但是由于 Word 自身没有开启 DEP，使得栈中的 shellcode 仍然能够得以执行。

另外，CVE-2010-2883、CVE-2011-0611、CVE-2007-4607 等样本展示了 Spraying 和 ROP 结合的攻击方式。通过喷射将攻击代码大量部署在内存中，能够有效突破内存地址随机化安全机制，同时利用 ROP 和 JIT 的方式来保证将攻击代码所在内存区域可执行。此种攻击方法可以同时绕过 DEP 和 ASLR 机制。

实验表明，本文的系统不但可以检测 ret-to-libc、SEH exploit 等传统利用攻击，也能成功检测到 APT 攻击所应用的 ROP、JIT Spraying、Spraying 结合 ROP 等方式的复杂漏洞利用攻击。

### 7.3 案例分析

本文挑选 CVE-2013-2551 和 CVE-2007-4607 2 个案例来对 ECfield 的检测过程和效果进行详细阐述。这 2 个漏洞分别利用了目前在 APT 等漏洞利用攻击中广泛使用的 Spraying 结合 ROP、JIT

**表 2** 样本程序分析结果

| CVE 编号 | 控制流转移地址 | 控制流转移指令 | 目的指令 | 目的地址 | 攻击类型 |
|---|---|---|---|---|---|
| CVE-2007-4607 | ntdll+0x465f7 | call ecx | cmp al, 0x35 | 0xc3f0101 | JIT Spraying |
| CVE-2010-3333 | ntdll+0x465f7 | call ecx | pop edi; pop esi; ret | msxml5+0x12890 | SEH exploit |
| CVE-2010-2883 | cooltype+0x8b308 | call [eax] | add ebp,0x794; leave; ret | icucnv36+0xcb38 | Heap Spraying 结合 ROP |
| CVE-2011-0611 | flash+0xaa7d7 | call [eax+0x8] | xchg eax, esp; ret | msvcr71+0x8b05 | Heap Spraying 结合 ROP |
| CVE-2012-0158 | mscomctl+0x48a56 | ret 0x8 | jmp esp | mscomctl+0x3c30 | ret-to-libc |
| CVE-2012-4969 | mshtml+0x25c4c0 | call [eax+0x8] | xchg eax,esp; ret | msvcr71+0x8b05 | Heap Spraying 结合 ROP |
| CVE-2013-2551 | mshtml+0x1bc545 | call [eax+0x8] | xchg eax,esp; ret | msvcr71+0x8b05 | Heap Spraying 结合 ROP |
| CVE-2014-1761 | mscomctl+0x14a34 | ret | add esp,0xc; ret | mscomctl+0x5e6ae | Heap Spraying 结合 ROP |

Spraying 攻击方式。同时，2 个漏洞程序在运行过程中都存在动态代码执行的情况。本文的系统能够准确地识别合法的动态指令代码场景，并对异常动态代码执行情况进行准确判定。

**案例 1**　CVE-2013-2551 漏洞利用攻击

此漏洞是由于 IE 8 浏览器没有对 dashstyle.array 的长度做出正确的验证，导致整数溢出。在本实验中，样本运行在 Windows 7 专业版系统中。

程序在运行时，共有 256 次执行动态生成代码的情况，其中包括执行 Hook 代码和 Flash JIT 代码等。ECfield 准确地识别了这些正常的动态代码执行的情况，没有产生漏报和误报。

程序的异常控制流转移发生在 mshtml 模块中，偏移为 0x1bc545 处的指令：call [eax+0x8]，目标地址是 0x7c348b05，该地址在 msvcr71.dll 中。由于 ECfield 在每一次间接函数调用时，会对控制流转移的合法性进行判断，此次调用的目标地址不是一个外部调用函数的起始地址，ECfield 将此次控制流转移判定为不合法。

通过 ECfield，可以进一步获悉此漏洞利用的攻击过程。msvcr71.dll 是加载到 IEXPLORE 进程中的非随机化模块，模块中地址 0x7c348b05 处开始的指令片段是：xchg eax, esp; ret；通过这两条指令来切换栈区域，栈顶变为 0x0c0c0c0c，并且通过 ret 返回到存放在新栈顶处的地址 0x7c341748，继续执行后续的 ROP 片段。从系统单步跟踪利用攻击点之后的指令序列可知，0x0c0c0c70 处存放了 shellcode，攻击者修改了所在页面的执行属性，最终跳转至 shellcode 执行。另外，ECfield 通过检测相邻的内存区域，发现它们不但具有相同的执行属性，同时相同偏移处的数据相似度达到 100%。由此判定，此漏洞利用攻击采用了 Spraying 结合 ROP 的攻击方式。

**案例 2**　CVE-2007-4607 漏洞利用攻击

此漏洞利用采用了 JIT Spraying 的方式进行攻击。攻击样本通过加载一段恶意的 Flash 视频文件，使得其中的 Flash 脚本运行，脚本将 shellcode 喷射到大量内存区域中，随后通过触发 IE ActiveX 插件的漏洞，促使控制流跳转至内存中某段 shellcode，进而实施攻击。此样本运行在 Windows 7 专业版系统中，Flash Player 的版本为 10.3.4。

由于在程序执行过程中 Flash 的 JIT 指令所在的内存区域具有可执行属性，所以通过 JIT Spraying 能够将嵌入到 Flash 中的 shellcode 直接喷射到大量的具有可执行属性的内存中，攻击者不用额外构造 ROP 链来绕过 DEP 保护，因此，攻击者可以通过此攻击方式，同时绕过 ASLR 和 DEP；而且由于 CFI 方法[2]无法处理动态执行代码执行的合法性，所以此攻击方式也能够绕过 Total-CFI[11]、binCFI[10] 提出的检测方法。

Flash JIT 代码是程序运行过程中生成的动态代码。在每一次控制流执行至 JIT 代码模块时，ECfield 确定此代码模块所在的内存区域，检测与之相邻的内存区域的可执行属性和相同偏移处的数据相似性。当高度相似时，进一步通过反汇编当前 JIT 代码模块中指令序列，检测是否有高度疑似 shellcode 的指令，以此判定 Flash JIT 代码相关的控制流转移的合法性。

在此案例中，ECfield 共发现 17 828 次正常的 JIT 代码相关的控制流转移，在每次提取相关 DGC 模块和 DGC 函数之后，ECfield 将它们加入到安全执行轮廓中。异常的程序控制流转移发生在 ntdll 模块偏移为 0x465f7 的指令 call ecx 处。此函数调用的目标地址本应为 SEH 链中的异常处理例程函数，但是实际目标地址却为 0x0c3f0101。0x0c3f0101 处数据是 Flash JIT 动态生成代码。该地址所在的内存区域为[0x0c3f0000 ~ 0x0c3fffff]。在与之相邻的内存区域[0x0c3e0000 ~ 0x0c3effff]和[0x0c400000 ~ 0x0c40ffff]中，相同偏移处的数据与当前内存区域中的数据相似度达到 100%。同时，0x0c3f0101 开始的汇编指令序列中，包含了访问当前进程 PEB、获得模块列表 LDR 地址和获得关键 API 内存地址等疑似 shellcode 指令序列。由于相邻内存区域中数据高度相似，并且当前目标指令序列具有疑似攻击指令，因此，ECfield 将之判定为异常的控制流转移。通过随后的单步跟踪发现，shellcode 头部有大量重复的 nop 和 cmp al, 0x35 指令。这些滑板指令保证了控制流转移至 shellcode 的成功率。

### 7.4　时间运行开销

本文利用攻击检测方法具有较小的运行时间开销。在原型系统 ECfield 中，测量在样本开始运行至检测到利用攻击发生时的时间间隔 $t$，同时，也在原生 QEMU 模拟器中运行同样的攻击样本，并测试这一时间间隔 $t'$，具体数据如图 4 所示。ECfield 平均运行时间开销约为原生 QEMU 系统的 2.1 倍。由此可见，本文的方法具

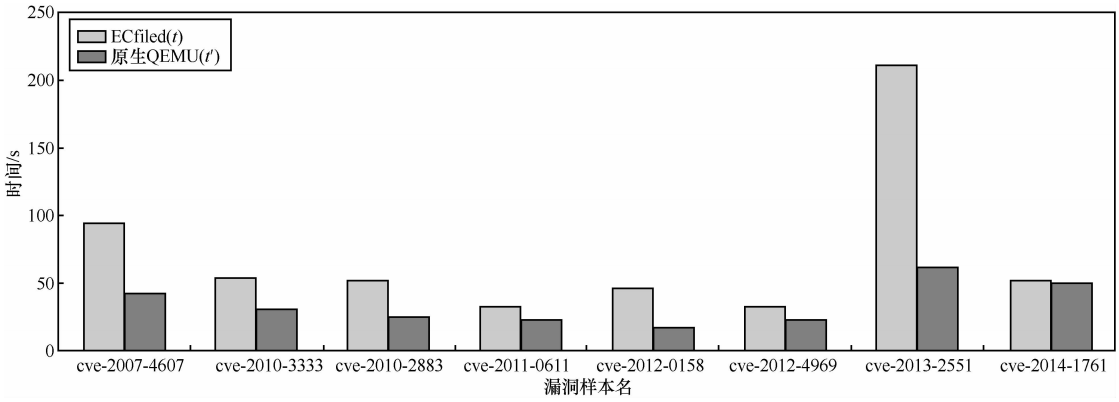


图 4　ECfield 与原生 QEMU 运行时间开销对比

有较高的利用攻击检测效率，能够用于漏洞攻击的实时检测。

在运行效率上，本文的实验系统比利用污点跟踪分析实现的分析系统，如 Argos 漏洞攻击检测系统[7]，具有较明显的优势。污点分析方法需要跟踪每一条指令来处理污点状态的传播过程。要达到理想的效果，分析粒度需要在字节级别，这导致运行时间开销大概增长 3~50 倍[4,5]。另外，Argos 仅采用直接数据依赖的污点分析，真实漏洞利用中还包含大量有关污点数据的间接数据流依赖和控制流依赖的情况，这使得这些基于污点分析实现的检测系统，很难保证检测效果的准确率。

## 8　局限性

本文方法对实际漏洞利用攻击具有较好的检测效果，但是也存在着一些局限性。首先，本文方法重点关注针对应用程序的漏洞利用攻击检测，无法检测发生在内核中的利用攻击。其次，在间接函数调用时，验证目标函数是否具有外部调用属性，对于某些 call-to-libc 类型的利用来说，如果 call 的目标恰好是 call 指令所在模块导入的函数（如 kernel32 模块中的 WinExec），并且直接通过此函数执行恶意行为，本文方法将认为此函数调用是正常的，造成漏报。由于 Windows 7 等操作系统中已经加入 SafeSEH、ASLR 和 DEP 等安全机制，攻击者很难仅通过一次 call 调用来绕过这些安全机制，完成所有的攻击步骤，所以此种利用方法很难应用在实际的漏洞攻击中。为了检测此种类型的攻击，可以重点关注如 WinExec、CreateProcess 等函数，通过在这些函数调用时检测函数参数是否包含恶意数据识别攻击。

## 9　结束语

本文提出一种基于异常控制流识别的漏洞利用攻击检测方法，能够在恶意攻击代码执行之前，检测到攻击发生。通过对二进制目标程序静态分析和动态执行监测，构建完整的安全执行轮廓，并限定控制流转移的合法目标。在函数调用、函数返回和跳转等控制流转移发生时，检测目标地址的合法性，将异常控制流转移判定为漏洞攻击，并捕获完整的攻击步骤。为验证本文方法的正确性，本文实现了基于异常控制流转移检测的漏洞利用攻击原型系统，并对若干实际高危漏洞进行实验。实验表明，本文的方法能够准确检测到利用攻击，并具备良好的运行效率，可以作为漏洞利用攻击的实时检测工具。

## 参考文献：

[1] Secunia[EBOL]. http://secunia.com/vulnerability-review/.2014.

[2] ABADI M, MIHAIBUDIU, ERLINGSSON U. Control-flow integrity[A]. Proceedings of the 12th ACM conference on Computer and Communications Security[C]. Raleigh, NC, USA, 2005.340-353.

[3] BOSMAN E, SLOWINSKA A, BOS H. Minemu: the world's fastest taint tracker[J]. Recent Advances in Intrusion Detection, 2011, 6961: 1-20.

[4] NEWSOME J, SONG D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[A]. Network and Distributed System Security Symposium[C]. San Diego, California, USA: Internet Society, 2005.

[5] SCHWARTZ E L, 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)[A]. IEEE Symposium on Security and Privacy[C].Oakland, CA, USA, 2010.317-331.

[6] FireEye[EB/OL]. http://www.fireeye.com/.2014.

[7] Argos[EB/OL]. http://www.few.vu.nl/argos/.2014.

[8] PORTOKALIDIS G, SLOWINSKA A, BOS H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation[J]. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006[C]. New York, NY, USA: ACM, 2006.15-27.

[9] YIN H, SONG D, EGELE M. Capturing system-wide information flow for malware detection and analysis[A]. Proceeding of the 14th ACM Conference of Computer and Communication Security[C]. Alexandria, VA, USA, 2007.116-127.

[10] ZHANG M W, SEKAR R. Control flow integrity for COTS binaries[A]. Proceedings of the 22nd USENIX Conference on Security 2013[C]. Berkeley, CA, USA, 2013.

[11] PRAKASH A, YIN H, LIANG Z K. Enforcing system-wide control flow integrity for exploit detection and diagnosis[A]. 8th ACM Symposium on Information, Computer and Communications Security[C]. Hangzhou, China, 2013.311-322.

[12] ZHANG C, WEI T, CHEN Z F. Practical control flow integrity & randomization for binary executables[A]. The 34th IEEE Symposium on Security & Privacy[C]. San Francisco, CA, USA, 2013.559-573.

[13] ZHANG C, WEI T, CHEN ZF. FPGate: The Last Building Block For A Practical CFI Solution[R]. Technical Report For Microsoft BlueHat Prize Contest, 2012.

[14] ROEMER R, BUCHANAN E, SHACHAM H. Return-oriented programming: systems, languages, and applications[J]. ACM Transactions on Information and System Security, 2012,15(1).

[15] DING Y, WEI T, WANG TL. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks[A]. Proceedings of the 26th Annual Computer Security Applications Conference[C]. New York, NY, USA: ACM, 2010.327-336.

[16] Heap FengShui[EB/OL]. https://www.blackhat.com/presentations/bh-europe-07/ Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.2014.

[17] BELLARD F. Qemu, a fast and portable dynamic translator[A]. Proceedings of the 14th USENIX conference on Security [C]. Baltimore, MD, USA, 2005.

[18] WANG MH, SU PR, LI Q. Automatic polymorphic exploit generation for software vulnerabilities[A]. 9th International Conference on Security and Privacy in Communication Networks[C]. Sydney, Australia. 2013.216-233.

[19] IDA Pro[EB/OL]. https://www.hex-rays.com/products/ida/,2014.

[20] Metasploit[EB/OL]. http://www.metasploit.com/,2014.

**作者简介：**

**王明华**（1986-），男，吉林长春人，中国科学院博士生，主要研究方向为二进制程序逆向分析、软件漏洞分析等。

**应凌云**（1982-），男，浙江永康人，博士，中国科学院高级工程师、硕士生导师，主要研究方向为恶意代码分析与移动智能终端安全。

**冯登国**（1965-），男，陕西靖边人，博士，中国科学院研究员、博士生导师，主要研究方向为密码学、信息安全。