计算机科学
COMPUTER SCIENCE

Python 虚拟机本地代码的安全性实证研究

蒋成满，华保健，樊淇梁，朱洪军，徐波，潘志中

引用本文

蒋成满，华保健，樊淇梁，朱洪军，徐波，潘志中. Python 虚拟机本地代码的安全性实证研究[J]. 计算机科学, 2022, 49(6A): 474-479.

JIANG Cheng-man, HUA Bao-jian, FAN Qi-liang, ZHU Hong-jun, XU Bo, PAN Zhi-zhong. Empirical Security Study of Native Code in Python Virtual Machines[J]. Computer Science, 2022, 49(6A): 474-479.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210600200

Python 虚拟机本地代码的安全性实证研究

蒋成满[1]　华保健[1]　樊淇梁[1]　朱洪军[2]　徐　波[3]　潘志中[1]
1 中国科学技术大学软件学院　合肥 230000
2 安徽信息工程学院　安徽 芜湖 241002
3 中国科学技术大学微尺度科学国家实验室　合肥 230000
(sa148@mail.ustc.edu.cn)

摘　要　Python 语言及生态是机器学习等人工智能系统的重要基础,已成为目前主流机器学习框架如 TensorFlow,PyTorch,Caffe,CNTK 等的首选实现语言。Python 虚拟机本身的安全性和可靠性对这些机器学习框架的安全性提供了基础保障,但 Python 虚拟机 CPython 内部包含大量由 C/C++构建的本地代码,其安全漏洞模式尚未被充分研究和理解,系统的漏洞分析和修复技术也亟待研究。为此,提出了一个对 Python 虚拟机本地代码的分析研究框架 PyGuard,该框架使用静态程序分析技术对虚拟机中的本地代码进行安全性扫描和分析;利用该框架对 Python 语言的官方虚拟机 CPython 进行了安全性实证研究,实验结果发现了最新版本的虚拟机(Cpython 3.9)中 45 个安全漏洞,表明了该框架对实际 Python 虚拟机本地代码安全性分析的有效性;基于该框架和安全性进行了实证分析,分析了虚拟机本地代码中的安全漏洞模式,给出了对安全漏洞的修复建议。

关键词:Python 虚拟机;本地代码;安全漏洞;程序分析
中图法分类号　TP311

Empirical Security Study of Native Code in Python Virtual Machines

JIANG Cheng-man[1],HUA Bao-jian[1],FAN Qi-liang[1],ZHU Hong-jun[2],XU Bo[3] and PAN Zhi-zhong[1]
1 School of Software Engineering,University of Science and Technology of China,Hefei 230000,China
2 Anhui Institute of Information Technology,Wuhu,Anhui 241002,China
3 Hefei National Laboratory for Physical Sciences at the Microscale,University of Science and Technology of China,Hefei 230000,China

Abstract　The Python programming language and its echo-systems continue to play important roles in modern artificial intelligent systems like machine learning or deep learning,and are among one of the most popular implementation languages in modern machine learning infrastructures like TensorFlow,PyTorch,Caffe or CNTK. The security of the Python virtual machines is critical to the security of these machine learning systems. However,due to the existence of huge native code base in Python's CPython virtual machine,it's a great research challenge to study the security vulnerability patterns in Python virtual machines and the techniques to fix these vulnerabilities. This paper presents a novel vulnerability analysis framework PyGuard,which makes use of the static program analysis techniques to analyze the security of native code in Python virtual machines. This paper also introduces a prototype implementation of this framework and reports the experimental results of an empirical security study of the CPython virtual machine (version 3.9):we have found 45 new security vulnerabilities which demonstrates the effectiveness of this system. We have conducted a thorough study of the vulnerability patterns and given a taxonomy.

Keywords　Python virtual machines,Native code,Security vulnerabilities,Program analysis

1　引言

Python 语言及其生态因编程便利、接口友好、易部署等优势,在机器学习、人工智能、数据科学和信息安全等领域中得到越来越广泛的应用。主流机器学习框架如 PyTorch,TensorFlow、Caffe、CNTK、飞桨等都提供了 Python 语言的编程接口支持。

目前机器学习系统已经被越来越多地应用在医疗、自动驾驶等高安全需求领域中,这些领域对机器学习系统以及 Python 语言提出了更高的安全需求。Python 语言是一种动态类型的解释型语言,其安全性研究可分成两个方面:一方面是对 Python 语言本身的安全性的研究;另一方面是对 Python 语言运行时环境即 Python 虚拟机的安全性的研究。对于第一个方面,由于 Python 程序是动态类型的,安全相关的操作在运行过程中都会被检查和处理,尽管这相对于静态类型的语言来说,降低了执行效率,但动态检查可以保证程序的类型安全和内存安全。

对于第二个方面即 Python 虚拟机的安全性由于实际的 Python 虚拟机如 CPython 等都包含大量的 C/C++构建的本地代码,Python 虚拟机本地代码的安全性,已经成为

基金项目:中国科学技术大学研究生教育创新计划项目(2020YCJC41)
This work was supported by the Graduate Education Innovation Program of USTC(2020YCJC41).
通信作者:华保健(bjhua@ustc.edu.cn)

Python 语言生态安全性的短板，也是易被攻击的薄弱环节；已经公布的许多 Python 安全漏洞都和 Python 虚拟机的安全性相关，例如，在 CVE-2017-1000158 漏洞中，整型溢出导致堆溢出，进而导致任意代码执行。

```
1. img_t table_ptr; /* struct containing img data, 10 kB each */
2. int num_imgs;
3. …
4. num_imgs=get_num_imgs();
5. table_ptr=(img_t*)malloc(sizeof(img_t)*num_imgs);
6. …
```

上面的代码(cwe-190)打算分配大小 num_imgs 表，但由于 num_imgs 变大，计算确定列表的大小最终将溢出。这将导致待分配非常小的列表。如果后续代码在列表上运行，认为它和 num_imgs 一样长，可能导致许多类型的边界问题。

我们认为，Python 虚拟机本地代码安全性研究的挑战，主要包括两个方面。

(1)本地代码安全性研究挑战：本地代码由于其语言 C/C++中包括的固有不安全特性，容易引入整型溢出漏洞、格式化字符串攻击、缓冲区溢出、堆溢出等安全漏洞且难以被发现和修复。尽管目前已有大量针对这些安全漏洞的一般性研究工作，但还没有专门针对 Python 虚拟机本地代码的特有安全漏洞及其分类的专门研究和系统性的结论。

(2)Python 代码和本地代码交互的安全性挑战：Python 代码可以通过本地方法接口与本地代码交互，由于托管代码和本地代码内存模型、数据表示、语义等方面的本质差异，使得这类交互中可能引入特有的安全漏洞，对这方面的安全漏洞的产生机理和修复机制的研究亟待进行。

针对以上安全研究现状和技术挑战，本文提出了一个针对 Python 虚拟机本地代码安全的扫描框架 PyGuard，在 PyGuard 中，我们采用了静态程序分析和代码扫描技术，并结合数据统计分析方法和差异代码测试技术，对 Python 虚拟机中的本地代码、以及 Python 托管代码的本地方法接口代码进行安全扫描。对于本地代码，我们充分采用了现有成熟的本地代码技术、方法以及工具；而对于更加特殊的 Python 本地方法接口代码，我们研究并给出了有针对性的代码扫描技术和方案，并进行了实现。

为了验证该方法和框架的有效性，我们实现了该框架 PyGuard 的原型系统，并利用该原型系统，对 Python 官方虚拟机 CPython 3.9 版本进行了系统安全性扫描和分析，该版本的虚拟机中共包括 423 721 行本地代码。实验结果表明，PyGuard 发现了 CPython 虚拟机中 45 个新的安全漏洞，这表明该系统对于扫描和发现实际 Python 虚拟机本地代码的安全漏洞是有效的。

本文对 PyGuard 扫描得到的 Python 虚拟机中的安全漏洞进行了系统性的研究分析，总结了这些漏洞的特点，并给出了对这些安全漏洞的修复建议；这给后续该方向的工作提供了有益的启示。

总结起来，本文的技术亮点和主要贡献如下：

(1)提出了一个针对 Python 虚拟机本地代码进行安全性分析的框架 PyGuard；

(2)构建了该框架的实际原型系统，并利用该系统对 Python 3.9 版本的虚拟机 CPython 中的本地代码进行了安全性扫描分析和实证研究；

(3)系统分析了 Python 虚拟机本地代码的安全漏洞特点，并进行了系统的漏洞分类，给出了对安全漏洞的修复建议。

本文第 2 节介绍 Python 虚拟机本地代码和本地方法接口代码安全性研究的相关工作；第 3 节介绍本地代码安全性分析研究框架 PyGuard 的架构设计；第 4 节介绍在 Cpython 3.9 虚拟机上进行的实验并对实验结果进行分析，以验证系统的有效性；第 5 节总结一般性的安全漏洞模式，并讨论修复建议；最后总结全文并展望未来。

2 相关工作

对本地方法代码和对虚拟机本地方法接口安全性的研究，一直是非常重要的研究方向，目前已经有较多相关的研究工作，本节对相关研究工作进行介绍。

2.1 本地方法代码安全性

CPython 虚拟机是使用 C 语言实现的 Python 解释器，然而 C 代码本质上是不安全的，许多研究也证实了这一点[1-2]。C 语言的安全性研究一直受到业界的关注。例如，Ccure[3] 和 Cyclon[4] 通过静态和动态检查的组合为 C 代码提供了安全保证，在保留 C 语言的语法和语义的同时，防止 C 程序中常见的缓冲区溢出、格式化字符串攻击和内存管理错误。

缓冲区溢出会导致存储单元被重写，引发不可预料的后果。Stack Smashing Protection 是一种简单而高效的缓冲区溢出保护技术，但其容易受到逐字节攻击。Wang 等[5] 提出多态 SSP(P-SSP) 技术，该技术的核心思想是为一个新的进程/线程或一个新的函数调用重新随机化 canaries，确保攻击者猜测 canaries 的优势不会随着不同的实验而累积，从而击败了逐字节攻击。而 Jang 等[6] 的研究则是通过生成替换代码的技术，检测和防止包含缓冲区溢出漏洞的语句。Bao 等[7] 将目标执行引入静态分析工具中，降低了缓冲区溢出静态检测误报率。Ren 等[8] 和 Dahl 等[9] 则引入机器学习和神经网络模型来检测缓冲区溢出漏洞。Shao 等[10] 则对缓冲区溢出分析领域的研究方向做了系统性的总结。

C 语言中内存错误引发了无数的安全问题。Zhang 等[11] 基于静态分析技术，提出一种基于路径敏感的值流分析内存泄漏检测方法。Gregory 等[12] 引入动态类型的 C/C++，通过在运行时使用动态检查每个对象的有效类型来检测内存错误。动态内存错误检测技术也被广泛应用于定位和修复导致内存错误的代码，Xu 等[13] 提出了一种基于动态二进制转换的动态内存错误检测方法，该方法结合调用堆栈跟踪方法和 IR 语言级内存错误检测方法，以提高检测成功率。但是这些技术不能处理细粒度的内存访问错误，并且大多依赖源代码或调试信息来恢复内存边界信息，不能直接应用于检测二进制代码中的内存访问错误。Li 等[14] 提出了内存访问完整性，一种在二进制可执行文件中检测细粒度内存访问错误的动态方法。Zhu 等[15] 就目前内存泄漏检测存在大量误报的原因，提出使用机器学习算法学习程序特征与内存泄漏之间的相关性，构建分类器，以降低内存泄漏检测结果的误报率。

2.2 本地方法代码安全性

现实世界的系统许多是异构的，它们使用不同编程语言开发，通过外部函数接口(Foreign Function Interface，FFI)进行交互。然而，由于语言特性的差异，如内存管理、异常处理和类型系统等，编写安全可靠的多语言程序并不容易。因此，针对 FFI 的研究也一直是业界的研究热点。

Furr 等[16]提出了一种多语言类型推断系统，用于检查 OCaml 外部函数接口调用，以防止引入类型和内存安全违规。他们的工作是跟踪 C 语言中的 OCaml 类型，以静态地防止 C 代码滥用 OCaml 类型，保证了 Ocaml 外部函数接口的类型安全性。并且进一步扩展了系统[17]，用于检查使用 Java 本地接口(JNI)程序的类型安全性。

Jiang 等[18]围绕 JNI 调用中内存泄漏的问题，扩展了 Java 字节码指令，消除跨语言分析的障碍，利用过程间分析方法检测 JNI 调用中的内存泄漏。Tan 等[19]提出 SafeJNI 的框架，通过对 C 代码进行静态和动态检查，保证了调用本地 C 方法时的类型安全。针对 Java 和本地方法中异常处理方式[20-22]的差异，构建了一个新的静态分析框架，用于发现 Java 本地代码中的异常处理错误。并在这些工作的基础上，进一步对 JDK 中的本地代码进行实证安全性研究[23]，提出了一些漏洞模式。

但是，对 Python/C API 安全相关的研究很少，主要集中在经典的漏洞模式和引用计数错误上。Python 使用引用计数来管理对象，但是，本地方法创建的对象不受垃圾收集器(GC)的控制，它的引用计数不能自动调整。Pungi[24]使用仿射抽象来静态分析程序的 SSA 形式。RID[25]使用了不一致路径对检查来放松 Pungi 的假设，提高了分析的准确性。Zhang 等[26]采用静态分析方法，揭示了 Python/C API 版本的演进情况，实现了一个工具链 PyCEAC，得到了不同领域的 7 个软件系统 Python/C API 的使用数据，同时总结了 Python/C 接口的安全风险，提出了漏洞模式。

3 系统架构设计与实现

本小节介绍 PyGuard 系统的架构设计与实现细节。PyGuard 的系统架构有两个重要的设计目标：

(1)PyGuard 的架构要对 Python 虚拟机具有普适性，即 PyGuard 要能够支持对不同 Python 虚拟机、以及同一款 Python 虚拟机的不同版本的安全性进行分析扫描，这样，我们不仅能够横向对比不同版本的 Python 虚拟机的安全性情况，还能纵向分析同一款 Python 虚拟机不同版本安全性演化的趋势。

(2)PyGuard 的架构要模块化地支持对不同安全特性的分析，许多安全漏洞是正交的，同时对许多安全漏洞的分析技术也在不断演进，引入模块化的支持，可以使架构更容易扩展。

基于以上设计目标，在图 1 中给出了 PyGuard 的架构。PyGuard 的架构共分为 3 个大的模块：输入处理模块、安全插件模块(核心分析引擎)、漏洞分析统计模块。

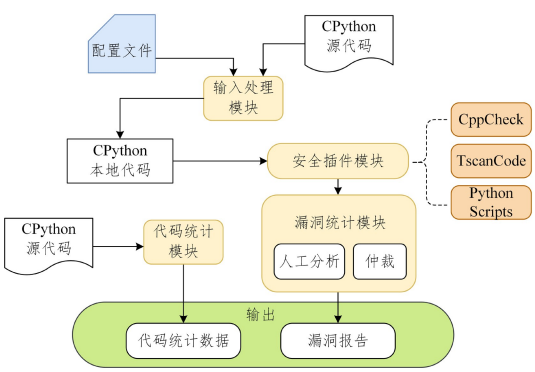

图 1 PyGuard 系统架构

Fig. 1 PyGuard system architecture

3.1 输入处理模块

输入处理模块的主要功能是提取 Python 虚拟机源代码中的本地代码和本地方法接口代码，该模块可以根据配置信息过滤目标目录。

在架构设计中把该模块独立出来，至少具有两方面的优势。首先，该模块将不同 Python 虚拟机以及不同版本的虚拟机的差异有效隐藏起来，例如，Python 官方虚拟机 CPython 中包括大量的本地代码以及本地方法接口代码，而基于即时编译的 PyPy 虚拟机使用 Python 作为实现语言，因此只包含本地方法接口代码；所有类似的这些差异都将在该模块中进行有效处理。其次，该模块向后续模块输出统一的中间表示，这可以简化后续模块的实现难度，降低系统复杂度。

3.2 安全插件模块

安全插件模块是 PyGuard 的核心模块，主要集成了一些现成的开源静态分析工具和我们构建的扫描插件。它接受输入处理模块提取的虚拟机本地代码作为输入，对特定的安全特性进行分析，并向后端输出漏洞扫描分析结果。

根据本文前面的讨论分析，Python 虚拟机本地代码中可能存在一般性的 C 代码安全漏洞、以及和 Python 本地方法接口特定的安全漏洞。

为了扫描 C 代码中常见的漏洞模式，例如缓冲区溢出、竞争条件和内存管理错误等，我们使用了 CppCheck 和 TscanCode 的组合。CppCheck 和 TscanCode 通过添加漏洞规则，支持检测空指针、数据越界、内存泄漏、运算错误等漏洞，TscanCode 准确率较 CppCheck 高。

Python 虚拟机中的一些错误是 Python/C 接口特有的，现有的检测工具不能扫描出这些错误，因此我们构建基于 Python 脚本的扫描插件来搜索识别这些模式中的漏洞。扫描脚本的构建过程如下：首先，系统研究并分析 Python 的本地方法接口规范，总结其中容易出现错误的所有漏洞模式；然后，基于这些漏洞模式，用 Python 基于过程内程序分析的技术，对漏洞模式进行扫描分析。

3.3 漏洞统计分析模块

漏洞统计分析模块负责对安全插件输出的漏洞扫描数据进行整理、统计、分析，并输出分析数据和分析结论。但此模块面临两个实现上的技术挑战：第一，鉴于静态分析工具和构建的安全插件都具有一定的误报率，该如何从扫描数据中分析得到真正的安全漏洞报告；第二，如何分析安全漏洞的漏报。为了解决第一个问题，我们采用了仲裁和人工分析确认相结合的技术，如果多个安全工具都报告了同一个漏洞，则该漏洞有较大概率得到确认；否则，该漏洞属于误报的概率较大，再结合人工分析的方式对其进行确认。为了解决第二个问题，我们使用了差异比较技术，即把正在被分析的 Python 虚拟机本地代码，与最新版的同一款虚拟机本地代码进行差异的比较，如果有安全漏洞在新版虚拟机中得到修复，则可确认这是系统的一个漏报。尽管从理论上看，由于新版本也未必修复了所有安全漏洞，因此这种判定技术不是完备的，但由于漏洞发现的不可判定性，实际上这也是非常有效的漏报率评测方案。

4 实验结果与分析

本节讨论对 PyGuard 系统进行的实验，并对实验结果数据进行讨论分析。实验的目标，是要回答两个重要问题：

(1)该系统是否有效,尤其是,该系统是否能够发现 Python 虚拟机中的未知安全漏洞;

(2)该系统的准确率,即误报率和漏报率如何,对多少以及哪些安全漏洞能够完成自动发现并能给出修复建议。

实验的平台基于 AMD Ryzen 7 3750H 架构、内存 4GB,运行 Ubuntu 14.04.4 版本操作系统。

4.1 实验目标 Python 虚拟机

选定的实验目标 Python 虚拟机 CPython 虚拟机(3.9 版本),首先 CPython 的 Python 官方虚拟机使用最广泛;其次,目前 Python 3.9 是最新正式版本的 CPython 虚拟机,之前版本的虚拟机中报告的漏洞多通过版本进行迭代修复,新版本的虚拟机的安全漏洞亟待检测和修复。但本文的分析框架和实验方法,同样适用于其它版本的 Python 虚拟机。

CPython 虚拟机完全用 C 语言构建,其主要代码结构、代码功能以及代码规模统计如表 1 所列。CPython 整体代码比较复杂,包含了 423721 行的本地代码,限于执行手工检查漏洞,我们将扫描重点放在 Modules(Python 的标准库的实现)、Objects(Python 内建对象的实现)、Parser(虚拟机词法分析和语法分析的实现)、PC(虚拟机编译支持文件)和 Python(虚拟机解释器核心引擎的实现)5 个模块的代码。

表 1 CPython 3.9 版本本地代码结构

Table 1 Cpython 3.9 native code structure

模块	功能	本地代码文件数	本地代码行数
Doc	源代码文档	0	0
Grammar	计算机可读的语言定义	0	0
Include	C 头文件	99	16442
Lib	Python 标准库文件	0	0
Misc	杂项	1	187
Modules	C 标准库文件	108	189728
Objects	Python 内建对象	42	88515
Parser	Python 解释器中的 Scanner(词法分析)和 Parser(语法分析)部分	19	5259
PC	Windows 编译支持文件	20	8985
Programs	Python 可执行文件和其他二进制文件的源代码	3	503
Python	Python 解释器中的 Compiler(编译器)和执行引擎部分核心	70	78001
Tools	用于构建或扩展 Python 的独立工具	0	0
总计		362	387620

4.2 安全漏洞数据及分析

4.2.1 安全漏洞数量统计分析

首先,我们使用 PyGuard 对目标目录的本地代码进行扫描,并对得的安全漏洞数量进行了分类统计,表 2 列出了扫描得到的每一类安全漏洞及其数量。

我们共计扫描得到了 Cpython 3.9 虚拟机中 45 个新的安全漏洞,这说明该系统对于扫描真实虚拟机的本地代码安全漏洞是有效的。我们将在下面的第 5 小节,针对整数溢出漏洞(22 个)、内存管理漏洞(14 个)、异常处理错误漏洞(6 个)和错误检查不足漏洞(3 个) 4 类安全漏洞模式进行深入分析,并展示具有代表性的例子,还将对一些错误模式提出修复建议。

表 2 CPython 3.9 版本的安全漏洞数量统计

Table 2 Security vulnerability statistics for Cpython 3.9

安全漏洞模式		安全漏洞个数
整型数溢出		22
异常处理错误		6
错误检查不足		3
内存管理漏洞	C 内存	4
	Python 内存	10
总计		45

4.2.2 准确率分析

静态程序分析对于检测和定位安全错误非常有效,从表 2 中可以看到,它发现了示例系统中的大量安全漏洞。但另一方面,受限于静态程序分析的表达能力,当前的静态分析工具都存在一定的局限性,尤其是在扫描大规模代码时,会报告大量的假阳性警告。下面分析实验中所使用的第三方静态分析工具以及我们开发的工具的准确率。

表 3 给出了两种静态分析工具以及我们自行开发的分析模块的准确性分析数据,包括总计的警告数、真实错误数以及假阳率。

表 3 静态分析工具的假阳率

Table 3 False positive rate of static analysis tool

静态分析工具	告警总数	真实错误数	假阳率/%
CppCheck	60	2	96.7
TscanCode	134	3	97.8
整形溢出	50	22	56.0
异常处理错误	290	6	97.9
错误检查不足	247	3	98.8
内存管理缺陷	194	10	94.8

从表 3 中可以看到,无论第三方分析工具,还是我们自行开发的脚本插件,假阳率都比较高。我们分析,静态分析的假阳率较高的原因主要有两个。第一,是受限于静态分析方法的表达能力,由于不运行程序的代码,因此静态分析方法只能对程序的可能执行情况做许多保守的估计,从而导致误报。这种现象不是分析 Python 本地代码的特有问题,例如,已有的针对 Java 虚拟机 JDK 的静态程序分析表明,静态分析工具的误报率也较高,到达了 97.5%到 99.8%。第二,由于第三方程序分析工具只是把 Python 虚拟机中的本地代码当成一般性的本地代码来扫描,因此,这类工具缺乏 Python 本地代码特殊性的领域知识,从而导致误报。例如,Python 虚拟机的本地方法代码接口,必须遵守许多额外的安全约束,而如果工具忽略了这些约束,会报告大量的警告。

为了从这些告警中分析得到真正的安全漏洞,我们进行了大量的手动分析验证工作,从告警中分析得到了真正的安全漏洞。将告警代码对比最新版本的虚拟机代码,若最新版代码修复了此处代码,则该告警是真实的安全漏洞。

从这些实验中可以得出,如果要进一步提高第三方工具

的准确度，可以考虑引入更精细的代码模型[27]，这是未来值得探索的方向之一。

5 安全漏洞分类及修复建议

基于前文讨论的安全漏洞分析数据，我们对这些安全漏洞进行了进一步的分析和分类，总结出具有一般性的安全漏洞模式，并讨论对这些安全漏洞的修复建议；以期对未来针对Python虚拟机本地代码漏洞扫描的工具构建提供指导。

5.1 整数溢出漏洞

C/C++本地代码中的整型数溢出漏洞主要包括算术溢出、整数值转换丢失（例如，将int型数值转换成short型，从而导致值被更改）和非法使用位操作，如移位（将C中的值移动至少与其位宽相同的位置等），这类安全漏洞在已有研究工作中已经被系统研究过。

Python虚拟机本地代码中，包括另一种Python特有的、用于接口代码中格式字符串表示类型转换，而导致的整型数溢出漏洞。在Python中，为支持Python整型和C/C++整型直接的相互转换，引入了PyArg_Parse，PyArg_ParseTuple和PyArg_ParseTupleAndKeywords等Python/C接口函数，这些函数使用了格式转换字符串，而这些格式字符串的误用会导致整型数溢出漏洞。表4列出了不安全的格式字符及其含义。

表4 不进行溢出检查的格式化字符

Table 4 Format characters without overflow checking

格式字符	Python类型	C类型
B	integer	unsigned char
H	integer	unsigned short int
I	integer	unsigned int
k	integer	unsigned long
K	integer	unsigned longlong

下面是Modules模块下socketmodule.c文件的一个整型数溢出的代码实例，代码使用不进行溢出检查的格式化字符'I'，将一个Python整型转化成一个C unsigned int无符号整型。

```
1. // cpython-3.9.0/Modules/socketmodule.c#1815
2. int port,cid;
3. …
4. if (!PyArg_ParseTuple(args,"II:getsockaddrarg",&cid,&port))
5.     return 0;
```

当Python端传入的整数值超过C unsigned int范围（0～4294967295），Python/C API不会引发溢出异常，而是直接进行截断，可能会导致致命错误或可利用的漏洞。

对这类整型数溢出漏洞的修复并不复杂，只需要对所涉及的格式字符进行语法制导的检查即可；对这类安全漏洞的修复可自动进行，这类漏洞可加入到漏洞库中。

5.2 内存管理漏洞

Cpython中C代码需要管理两个内存区域，C内存区域和Python内存区域，它可能会对这两个内存区域管理不当。

C语言提供了malloc和free等标准库函数，为程序员提供手动管理内存的能力，它一直是编程缺陷和安全漏洞的根源。由于手动管理内存，因此可能存在悬空指针引用、多次释放和内存泄漏等内存管理安全漏洞。我们使用CppCheck和TscanCode来识别与目标目录中的内存管理相关的缺陷，手动检查了大量警告，发现了3例内存泄漏和1例空指针解引用的情况。

比如，下面代码片段是PC模块下用于distutils的Windows Installer程序，给出了一个内存泄漏的安全漏洞实例。第2行通过fopen打开logFile，在11行返回之前应该先把logFile关闭，否则会造成文件所占用的资源泄漏以及下次访问文件时出现问题。

```
1. // cpython-3.9.0/PC/bdist_wininst/install.c#2542
2. logfile=fopen(argv[2],"r");
3. if (!logfile) {
4.     MessageBox(NULL,"could not open logfile",NULL,MB_OK);
5.     return 1; /* Error */
6. }
7.
8. lines=(char **)malloc(sizeof(char *) * lines_buffer_size);
9. if (!lines)
10. ++ fclose(logFile);
11. return SystemError(0,"Out of memory");
```

除此之外，Python/C接口还提供另外3类内存分配器：PyMem _ RawMalloc/PyMem _ RawFree，PyMem _ Malloc/PyMem_Free和PyMem_MALLOC/PyMem_FREE，用于Python堆分配和释放内存。其中PyMem_RawMalloc/PyMem_RawFree和PyMem_Malloc/PyMem_Free是Python私有堆空间的内存分配器，而PyMem_MALLOC/PyMem_FREE是CPython提供的宏，用于直接调用内存分配器。

```
1. // cpython-3.9.0/Objects/memoryobject.c#824
2. static int mbuf_copy_format(_PyManagedBufferObject * mbuf,
   const char * fmt)
3. {
4.     if (fmt !=NULL) {
5.         char * cp=PyMem_Malloc(strlen(fmt)+1);
6.         if (cp == NULL) {
7.             PyErr_NoMemory();
8.             return -1;
9.         }
10.        mbuf->master.format=strcpy(cp,fmt);
11.        mbuf->flags|= _Py_MANAGED_BUFFER_FREE_FOR-
           MAT;
12.    }
13.
14.    return 0;
15. }
```

上面的代码展示了Cpython 3.9中一个内存溢出的例子。函数功能是从可能会消失的基础对象中复制格式字符串。第5行通过PyMem_Malloc在Python内存区域给变量cp分配了strlen(fmt)+1的内存，在函数结束前并没得到释放，导致该部分内存被泄漏。

尽管目前已有针对Java和C等语言内存管理相关漏洞的实证研究工作[28-29]试图了解内存管理缺陷的性质、表现方式以及如何修复，但手动代码检查和手动运行时检测仍然是内存管理缺陷检测的主要方法。

5.3 异常处理错误

CPython提供PyErr_SetString，PyErr_SetObject批API函数来引发Python异常。然而，Python异常处理机制与

Python/C API 异常处理机制之间不匹配。在 Python 代码中抛出异常时,Python 解释器将立即终止它,并将控制传递给最近的 try 语句,并匹配异常类型进行异常处理。使用 Python/C API 引发的异常不能立即终止本机方法的执行,必须显式返回以避免意外的控制流。

```
1. // cpython-3.9.0/Objects/listobject.c#2412
2. if (self->allocated != -1 && result != NULL) {
3.   /* The user mucked with the list during the sort,
4.    * and we don't already have another error to report
5. */
6.    PyErr_SetString(PyExc_ValueError, "list modified during
      sort");
7.   result = NULL;
8. }
```

针对上面示例代码的修复很简单,只需在抛出异常语句之后加上 return 语句。但是,当涉及到函数调用时,情况就变得复杂了。比如一个 C 函数 f,调用另一个 C 函数 g,函数 g 在错误发生时抛出异常,函数 f 必须显式地处理调用 g 的两种情况:成功和异常的情况。如果处理不当,可能会导致异常处理错误。当 C 函数 f 调用 Python 方法时,情况会变得更加复杂。

5.4 错误检查不足

Python/C 接口使用两种不同机制来处理被调用函数中出现的错误:使用显式抛出异常的 API,或者使用代表特定含义的错误返回值。对前者的检查并不困难,但对后者的检查比较困难,这主要是由于特定的错误返回值和正确的返回值并无明确的区别。因此,程序代码容易忽略对这类值的强制检查,导致出现安全漏洞。

例如下面代码返回列表 consts 中,位置索引为 arg 的对象 constant,但是 constant 可能为空,在增加对象 constant 的引用计数之前必须先检查对象为非空。

```
1. // cpython-3.9.0/Python/peephole.c#150
2. PyObject *constant = PyList_GET_ITEM(consts,arg);
3. ++ if (constant != null) {
4. Py_INCREF(constant);
5. PyTuple_SET_ITEM(newconst,i,constant);
6. }
```

针对错误检查不足这类漏洞的修复,主要工作是在使用之前增加对函数返回值的显式检查。对这类漏洞的自动修复工作将是我们下一阶段的研究工作之一。

漏洞分类展示了 Python 虚拟机本地代码中最频繁出现的漏洞模式,这些信息不但对研究针对这些漏洞的程序分析方法和构建自动分析工具很有价值,对 Python 虚拟机的实现者在实现中避免这些安全漏洞也很有帮助。

结束语 针对 Python 虚拟机本地代码的安全性漏洞的研究面临的挑战,本文提出了一个对 Python 虚拟机本地代码进行安全漏洞扫描分析的框架 PyGuard,并给出了系统的原型实现。实验结果表明,该系统发现了 Python 官方虚拟机 Cpython 3.9 版本中 45 个新的安全漏洞,这证明该系统对于发现实际 Python 虚拟机中的安全漏洞是有效的,本文对这些安全漏洞进行了分析并给出了漏洞修复建议。

基于本文的工作,还有几个重要的研究方向值得进一步探索,本文做以下 3 点展望:

(1)基于机器学习的漏洞建模和漏洞发现技术越来越重要,基于机器学习技术扫描分析 Python 虚拟机中的本地代码漏洞值得研究;

(2)本文主要使用了程序分析的静态代码扫描技术,还可以进一步探索基于动态扫描,如符号执行[30-31]、污点分析等的 Python 虚拟机本地方法漏洞分析技术,并对可能的性能代价进行深入分析和评价;

(3)本文主要基于手工的方式完成了安全漏洞的修复,尽管这种方式精确度较高,但成本较高,后续可进一步探索全自动的安全漏洞修复机制[32-35]。

参 考 文 献

[1] KOOPMAN P, DEVALE J. The Exception Handling Effectiveness of POSIX Operating Systems[J]. IEEE Transaction Software Engineering, 2000, 26(9): 837-848.

[2] MILLER B P, FREDRIKSEN L, SO B. An Empirical Study of the Reliability of UNIX Utilities[J]. Communications ACM, 1990, 33(12): 32-44.

[3] NECULA G C, CONDIT J, HARREN M, et al. CCured: type-safe retrofitting of legacy software[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005, 27(3): 477-526.

[4] JIM T, MORRISETT J G, GROSSMAN D, et al. Cyclone: A Safe Dialect of C[C]// USENIX Annual Technical Conference. General Track, 2002: 275-288.

[5] WANG Z L, DING X H, PANG C B, et al. To Detect Stack Buffer Overflow with Polymorphic Canaries[C]// DSN. 2018: 243-254.

[6] JANG Y S, CHOI J Y. Automatic Prevention of Buffer Overflow Vulnerability Using Candidate Code Generation[J]. IEICE Transactions on Information and Systems, 2018, 101-D(12): 3005-3018.

[7] BAO T Y, GAO F J, ZHOU Y, et al. Automatic Verification of Static Buffer Overflow Alarm Based on Target Guidance Symbol Execution[J]. Journal of Cyber Security, 2016, 1(2).

[8] REN J D, ZHENG Z Q, LIU Q, et al. A Buffer Overflow Prediction Approach Based on Software Metrics and Machine Learning [J]. Security and Communication Networks, 2019(1): 1-13.

[9] DAHL W A, ERDODI L, ZENNARO F M. Stack-based Buffer Overflow Detection using Recurrent Neural Networks[J]. arXiv: 2012.15116, 2020.

[10] SHAO S H, GAO Q, MA S, et al. Research Progress of Buffer Overflow Vulnerability Analysis Technology[J]. Journal of Software, 2018, 29(5).

[11] ZHANG J, HUANG Z Q, SHEN G H, et al. C Program Memory Leak Mechanism Analysis and Detection Method Design[J]. Computer Engineering & Science, 2020, 42(5).

[12] DUCK G J, YAP R H C. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++[J]. arXiv: 1710.06125, 2017.

[13] XU H, REN W, LIU Z M, et al. Memory Error Detection Based on Dynamic Binary Translation[C]// ICCT. 2020: 1059-1064.

[14] LI W J, XU D P, WU W, et al. Memory access integrity: detecting fine-grained memory access errors in binary code[J]. Cybersecur, 2019, 2(1): 286-303.

(下转第 653 页)