Security bugs in embedded interpreters

阅读报告

江梦　P14206009

本文主要对目前解释器实现中存在的各种 bug 及其对系统安全的影响进行了分析，为扩建现有解释器或开发新的更安全的解释器提供了理论支持。

现在很多系统都提供了用户定制功能。出于灵活性和可移植性的考虑，系统通常定义一个字节码形式的指令集并在内核实现一个嵌入式解释器，解释器将提交到内核的字节码翻译为机器代码使其在内核中执行，从而获得更高性能。典型的例子就是 Berkeley Packet Filter (BPF)：操作系统内核接收用户空间指定的字节码形式的 BPF 过滤器，并实现一个内核解释器将其翻译为机器码执行，从而挑选应用程序感兴趣的包。然而嵌入式解释器可能会对系统安全造成危害，主要原因有三点：一是许多系统不使用沙盒技术，如进程隔离或软件故障隔离，因此解释器的漏洞可能导致系统被攻击；二是嵌入式翻译器经常会确认不可信的字节码，容易造成错误；三是字节码通常会接收输入数据，字节码和输入数据都可能是不可信的，使攻击者可以利用的空间更大。基于此，本文观察研究了嵌入式解释器可能产生的各种错误以及对系统安全的影响。

interpreter	arith.	br.	loop	func.	ext.	JIT	bytecode source	input source	description
INET-DIAG		Y					user space	kernel	network monitoring [13]
BPF	Y	Y				Y	user space	network	packet & syscall filtering [7, 15]
AML	Y	Y	Y	Y	Y		firmware	device	power management [11]
Bitcoin	Y	Y					network	network	digital currency [1, 16]
ClamAV	Y	Y	Y	Y		Y	file & network		antivirus engine [30]
TrueType	Y	Y	Y	Y			file & browser		font rendering [26, 27]
Type 2	Y	Y		Y			file & browser		font rendering [12]
UDVM	Y	Y	Y	Y			file	network	universal decompressor [19]
RarVM	Y	Y	Y	Y	Y		file	file	decompressor filter [17]
Pickle				Y	Y		file & network		data serialization [18]

图 1：嵌入式解释器及其特征总结

作者首先介绍了嵌入式解释器在系统中的各种应用方式。目前各种系统中，解释器已经有了广泛的应用，但其实现的主要目的还是为了能够执行解决特定问题的字节码。文中总结了多种解释器，并从算数操作、条件分支、向前跳转、用户定义功能、外部功能调用以及是否有 JIT 实现等方面分析了它们的特征。典型的解释器主要有：

（1）Linux Socket Monitoring Interface (INET-DIAG)，用于套接字监控。它支持比较和向前

跳转，不支持向后跳转以避免循环，因此解释器必须验证每个跳转位移为正向的从而排除无限循环。

（2）Berkeley Packet Filter (BPF)，应用于多种类 Unix 操作系统内核，主要功能是过滤链接层的数据包。Linux 支持用 BPF 进行系统调用的过滤，并提供了一个 BPF JIT 编译器。BPF 使用了包括两个寄存器和一个暂存的虚拟机，支持整数算术操作、逻辑操作、分支和向前跳转。典型的错误主要有造成系统崩溃的除零，以及使用 BPF 前未将其占用的内存区域清零而造成的信息泄露。

（3）ACPI Machine Language (AML)，定义了 ACPI 控制方法，用于指示操作系统内核如何处理电源管理事件。由于 AML 字节码通常加载于硬件，有高可信度和较高的权限进行各种操作，而 Linux 内核允许用户空间应用程序重写某些控制方法，给攻击者提供了在内核中注入并执行攻击性代码的缺口。

（4）Bitcoin，使用一个栈结构的解释器来定义网络事务。

（5）ClamAV 防毒工具使用基于 LLVM 的字节码表示多种病毒软件的特征。ClamAV 的解释器和 JIT 对字节码沙盒使指针不会越界、没有无限循环以及禁用外部功能调用。

（6）TrueType and Type 2 Charstring，TrueType 定义了一种描述字体的隐式语言，支持算术逻辑操作、分支、循环以及功能调用，Type 2 Charstring 与它相似但不允许跳转和循环。

（7）Universal Decompressor Virtual Machine (UDVM)，用于包分析器 Wireshark，可以解压缩某些流协议。UDVM 提供了 64KB 的内存和调用栈，支持可直接寻址的算数、逻辑和字符串操作，以及分支和跳转。

（8）RarVM.，是一个类 x86 的寄存器机。用于 RAR 文件使用 RarVM 字节码实现一些输入数据的可逆冗余变换。RarVM 与 ClamAV 类似，但允许部分外部功能调用。

（9）Pickle。Python 标准库提供了 Pickle 模块用于序列化和反序列化 Python 对象。Pickle 库的错误使用会导致攻击权限提升或在远端机器执行的攻击代码。

为确保系统安全性，上述解释器都应该能拒绝不可信的字节码并对有不可信输入的字节码采取防护措施。很多错误就是由于未能完全实现这个原则而产生的，作者对这些在解释器中发现的漏洞进行了研究分析，主要有以下几类：

（1）资源枯竭（Resource exhaustion）：若字节码支持跳转，可能出现向后跳转导致无限循环。类似地，调用子程序也可能引起使栈溢出的无限递归。避免无限循环或递归的一种常见方法是限制执行的指令数和嵌套函数调用的层数。但是很多设置了相应计数器的复杂解释器仍然存在这种错误。

（2）算术错误（Arithmetic errors）：支持算术操作的解释器很容易出现算术错误，多数是由预期之外的目标机算术行为引起的，如除 0 或值溢出等，但是对这种算术错误的检查很复杂且易于出错。

（3）信息泄露（Information leak）：嵌入式解释器可能无意识的将其输入或执行环境暴露给字节码，从而使攻击者可以提取信息和观察执行结果。这种情况通常发生于提供了寄存器、暂存或者栈用于执行字节码的解释器。若这些存储区域未能被解释器正确初始化，那么攻击者就可以获得里面存储的有关系统或其他用户的敏感信息。

（4）攻击代码执行（Arbitrary code execution）：外部调用会破坏解释器和主系统直之间的隔离，可能导致系统中执行攻击性代码。若不能保证字节码及其输入都是可信的，解释器应该确保字节码与系统隔离，以防止攻击代码对系统的危害。

（5）内存污染（Memory corruption）：由于许多嵌入式内存器的编写语言并不安全，可能存在程序错误导致内存污染。例如忽略了字符串操作的边界检查而造成的栈的不当处理。

（6）JIT 喷射（JIT spraying）：一些嵌入式解释器采用了 JIT 工具将字节码编译为本级代码以加快执行速度，而由 JIT 生成的代码会引起一种新的攻击方式——JIT 喷射，即攻击者将 shell 代码编码为常量写在外表良好的字节码中，这些常量会造成 ROP（return-oriented programming）攻击。例如当特殊构建的 BPF 字节码编译加载到内核时，攻击者可以触发内存污染并跳转到被 JIT 编译的代码中，其中存在已经编译为机器指令的攻击者可以控制的常量。JIT 喷射的产生原因有两个：一是 JIT 编译的字节码通常驻留在可以同时写入和执行的页中，导致现有的保护技术失效；二是 JIT 工具更可能编译来源不可信的字节码。

嵌入式解释器所造成的错误繁多复杂，而且多数对系统安全有巨大的危害。因此，设计并实现一个安全的嵌入式解释器至关重要。作者提出了一些提升嵌入式解释器的安全性的指导方针：

（1）进程隔离：在不以性能为关键因素的情况下，可以通过进程隔离来保证解释器安全性。例如网页浏览器 Chrome 将不同页面的 JavaScprit 运行隔离在操作系统的不同进程中。但是这种方法有一定局限性：首先，由于 IPC 使性能降低，不适用于有高性能要求的应用程序；其次，它不能避免语义错误，解释器仍可能生成语义不一致的错误代码；最后，进程隔离很难应用于操作系统内核中。

（2）有限的资源消耗：运行在主机系统中的解释器必须确保字节码的执行不会耗尽系统资源，尤其对提供跳转和子程序的解释器，应该监控并限制运行时间和内存使用以避免无限循环。像 INET-DIAG 和 BPF 就采用了只允许向前跳转或限制执行指令的执行时间以约束解释

器对资源的使用。

（3）有限的特征集：一个有趣的规律是，字节码的表现能力越强，相应的解释器的实现就会包含越多的不变量，于是给攻击载体提供了广阔空间。嵌入式解释器的设计者应该注重灵活性和安全性二者之间的平衡。例如 INET-DIAG 不支持算数操作，就不容易产生如除 0 等算数错误。

（4）有限的系统调用：字节码程序与外部联系或处理主机发来的输入时，解释器应该定义在它们之间定义一个明确的借口。例如，BPF 字节码有清晰的与主机间的输入和输出接口，输入是一个包而输出是一个表明该包是否被过滤的位。

总之，本文分析归类了嵌入式解释器中存在的各种安全漏洞及其对系统的影响，并提出了一种降低普通漏洞的方法。文章主要贡献如下：（1）研究了各种系统中对嵌入式解释器的实际应用；（2）分析了嵌入式解释器存在的各种错误以及相应的攻击方式；（3）在对嵌入式解释器及其错误研究的基础上，总结了目前最先进的防御技术并提出了保证的安全性的方针。解释器开发人员可以通过借鉴此文的研究结果，了解各种解释器可能出现的错误，并结合作者提出的安全性建议构建解释器安全策略，从而设计更安全的新型解释器。

但是，本文的研究还存在一定不足之处。作者利用了 KLEE 和 SAGE 测试工具探索代码路径并构建输入数据来发现解释器中存在的错误。但是测试工具并不能完全高效地发现所有 bug，而且在测试时如何构建不可信的字节码和不可信的输入数据也十分重要。作者对于现有解释器 bug 的具体测试细节没有给出相关描述，我认为应该在解释器错误部分给出一些相关测试实例，以便更好的阐述解释器中错误的来源和它对系统的危害性。其次，由于从头构建一个解释很容易出现错误，通常是通过重用已有的字节码格式以及在已有解释器基础上扩展重建。开发人员通常选择一套组件来构建或定制解释器，组件构成了解释器的相关功能。例如，用于进程隔离的一种安全机制——Seccomp 系统，就是重用了 BPF 解释器来执行特定的系统调用过滤规则。基于此，我认为作者可以在对解释器的介绍部分详细描述各种现有解释器的优势和问题，以便阅读者能够有更直观的印象。本文研究的下一步工作可以是研究现有解释器出现各类错误的几率，以及避免漏洞机制的效率。