

基于 GCC 插件的堆栈保护技术研究

王亚刚, 曹耀彬, 王 萌

(西安邮电大学 计算机学院, 陕西 西安 710121)

摘 要: 缓冲区溢出攻击是计算机中最主要的漏洞之一, 在所有的缓冲区溢出中主要攻击的区域是程序的堆栈, 为了改进 GCC 编译器对于基于字节猜测的堆栈溢出防护的不足, 在 GCC 提供的最新插件基础上, 提出了一种堆栈保护增强插件, 并将其作为 GCC 编译器后端的一个编译优化过程. 实验结果表明, 新增的编译优化遍, 在一定程度上增加了攻击者对 canary 暴力破解的难度, 有效地缓解了 GCC 编译器堆栈防护的不足.

关键词: GCC 编译器; GCC 插件; 堆栈溢出; 软件安全

中图分类号: TP31

文献标识码: A

文章编号: 1000-7180(2018)01-0133-04

Research on Stack Protection Technology Based on GCC Plugin

WANG Ya-gang, CAO Yao-bin, WANG Meng

(School of Computer Science, Xi'an University of Posts and Telecommunications, Xi'an 710121, China)

Abstract: Buffer overflow attack is one of the main vulnerabilities in the computer. In all areas, the most likely to attack is the program stack. In order to enforce the GCC stack overflow protection, which is not enough to defense the byte guess attack. On the basis of the latest plug-in provided by GCC, a stack protection enhancement plug-in is proposed and used as a compilation optimization process for the back end of the GCC compiler. The final experimental results show that the new compiler optimization pass, to some extent, increasing the difficulty of the attacker to guessing the canary, which using byte for byte brute force attack. And the plugin can effectively alleviate the shortage of the GCC stack smashing protection.

Key words: GCC compiler; GCC plugin; stack overflow; software security

1 引言

大部分蠕虫病毒都是基于堆栈溢出漏洞^[1]而设计实现的, 其将预先设置好的代码注入到主机中, 其中著名的“Code Red worms”与“Slammer worm”病毒就是利用溢出漏洞实施攻击的代表. 蠕虫病毒最早要追溯到 1988 年 11 月, 当时许多机构不得不切断和互联的连接, 因为在互联网中出现了首例蠕虫病毒“morris worm”. 首例蠕虫病毒的出现给当时不太成熟的计算机网络造成了很大的破坏, 同时也引起了人们对这种漏洞的重视, 即堆栈溢出攻击.

堆栈溢出存在于各种程序中, 防止堆栈溢出攻击最有效的办法是数组边界检查^[2], 边界检查^[3]能够有效阻止所有的缓冲区溢出漏洞. 但并不是所有

的堆栈溢出都是漏洞, 只有那些被恶意代码利用的, 改变程序正常的执行流程的才属于溢出漏洞. 编程语言 Java, C#, shell 的运行环境能够探测堆栈溢出, 发现溢出时会产生异常信息. 虽然 Java 这类语言编写的程序不会出现堆栈溢出, 不过 Java 的运行环境虚拟机是 c 或 c++ 编写的, 所以其虚拟机中也存在堆栈溢出漏洞, 因此光靠语言本身不能解决堆栈溢出问题^[4]. 基于此, 本文从编译的角度出发分析了 GCC 编译器中的堆栈保护策略, 由于攻击的多样性与复杂性, GCC 编译器中堆栈保护对一些巧妙设计的攻击行为是防不胜防.

2 GCC 的堆栈保护

GCC 编译器并没有实现 StackGuard^[5], 而是实

现了其增强版本的 SSP (stack smashing protection), SSP 又称为 Propolice, SSP 设置的堆栈溢出保护主要涉及以下三个方面:

- (1)重新设置本地变量的位置,而不是根据定义的顺序;
- (2)将函数栈指针复制到本地变量的前面;
- (3)最后将 canary 插入函数栈帧中,保护函数的返回地址和栈指针.

当调用 GCC 编译器编译程序时,需要指定一些特定的编译选项,当在 GCC 命令行选项中加入 `fstack-protector` 时, GCC 会调用 SSP 在函数堆栈中设置堆栈溢出探测保护值 canary 值,在图 1 中可以看到设置堆栈保护与无堆栈保护汇编指令的差别,首先在当前函数执行压栈指令时,即在 `stack prologue` 中主要多出了 `movq %fs:40,%rax` 与 `mov %rax -8(%rbp)` 这两条汇编指令,其中 `%fs:40` 处存储的是系统设置的 canary 值,这两条指令的作用是将 SSP 设置堆栈溢出探测 canary 压入当前栈帧中;中间部分指令是存储当前函数堆栈中的本地变量;最后当函数返回执行出栈操作时,即在 `stack epilogue` 中,左边使用编译选项 `-fstack-protector` 产生的编译指令,在函数栈帧退出前使用 `movq -8(%rbp),%rax` 与 `xorq %fs:40,%rax` 这两条汇编指令将函数栈帧设置的 canary 与原值进行比较,如果不一致则说明当前函数栈帧存在堆栈溢出行为,因为攻击者恶意修改了堆栈中的 canary 探测值,所以此时程序不会正常终止,而是调用 `__stack_chk_fail` 堆栈检查出错函数,警告用户函数执行过程存在数据越界行为.

上述设置堆栈保护与无堆栈保护建立的函数栈帧的对比如图 2 所示,从图中可以看出 SSP 设置有无函数堆栈结构有很大的区别,本地变量的 i, j, k 的存储顺序不是按定义的顺序; `buf[8]` 字符数组移动到变量的前面与 canary 值的后面,这样不管 `buf` 是否溢出都不会破坏到本地变量 i, j, k , 因为堆栈的增长方向与内存的增长方向正好相反;在函数栈帧中多出了 canary 字段.

不过 GCC 编译器中 SSP 只能检查到堆栈是否被破坏,不能够防止堆栈溢出. 如果堆栈上的 canary 值被正确的值覆盖,那么这种探测将很容易绕过 SSP 保护. 当前的网络服务架构 `forking` 或 `pre-forking` 应用在许多服务器程序中,例如 Apache, Samba, Lighttpd, 在一些桌面应用程序如浏览器 Firefox 与 Chrome, 以及 Android 的 Zygote 也有广泛的

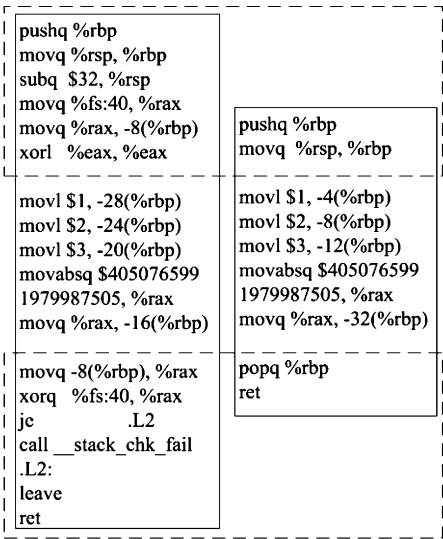


图 1 汇编指令对比图

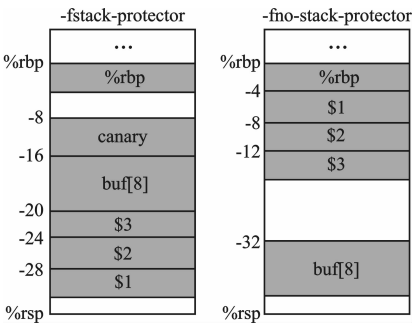


图 2 函数压栈对比图

应用. 所以目前攻击者都非常倾向于使用暴力破解的方式突破 GCC 编译器的堆栈保护 SSP. Fork 函数系统调用时 SSP 在父进程函数栈帧中设置的 canary 也会一同被复制到所有新创建的子进程中, 所有由父进程衍生出来的子进程的栈帧的 canary 都是一样的. 基于 `fork` 创建新的进程的网络服务程序存在非常危险的 `byte-for-byte` 攻击漏洞. 对于 canary 值, 如果每次猜测 canary 值中的一个字节, 那么攻击者最多只需要尝试 256 次 (0~255 可能的取值), 基于 `byte-for-byte` 的攻击首先会猜测 canary 值的第一个字节, 如果 GCC 的 SSP 探测到了溢出, 那么该连接会立马终止, 如果程序正常执行则说明猜测的第一个字节的值是正确的. 每个字节最坏的情况下攻击者只需要猜测 256 种情况, 第一个字节被顺利攻破以后, 那么攻击者就可以按照图 3 中的步骤——去猜测剩下 7 个字节 (64 位系统), 这样 64 位系统的 canary 的八个字节在很短的时间内就会被攻击者突破.

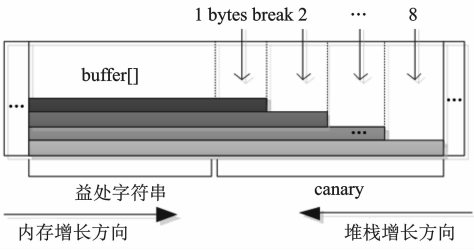


图 3 byte for byte 攻击探测 canary

3 堆栈保护增强插件

本文设计的 GCC 插件^[6]plugin for SSP 最关键的是函数栈帧 canary 地址记录表的实现,对于调用 fork 系统服务后新创建的子进程,为了实现所有其栈帧中 canary 值的更新,plugin for SSP 设计了一种记录 canary 值的地址记录表,如图 4 所示,该表被放在堆内存区域,地址记录表的每一个表项存储一个栈帧中 canary 值的地址,栈帧 A 与栈帧 B 在 canary 地址应用表中都有其对应的堆栈上设置的 canary 值的存储地址. 这样程序有了这些信息,在调用 fork 函数后,就能够轻而易举找到新创建的子进程上下文中从父进程继承而来的 canary 探测值,进而可以重新初始化 canary 探测值,实现系统中不同进程堆栈中 canary 探测值的多样化.

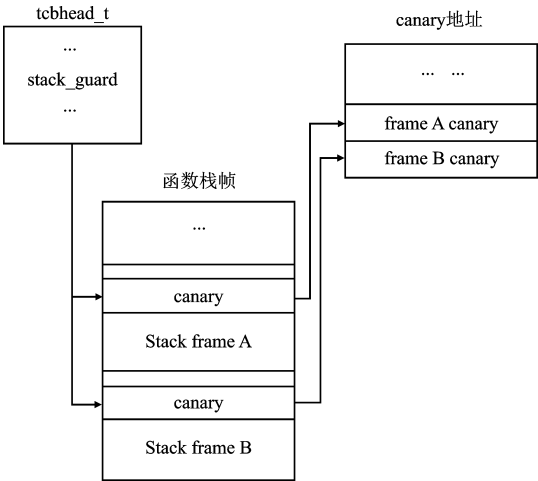


图 4 canary 地址记录表的设计

在调用 fork 函数时相应的 canary 地址记录表也要被拷贝到子进程中. 父子进程对于新建立的函数栈帧,如果编译的程序中设置了堆栈保护,那么就要在父子进程相应的 canary 地址记录表中增加相应的表项,保存其堆栈上设置的 canary 值的地址.

当 GCC 编译器开始执行后,很重要的一步是解析命令行编译选项等参数信息. 在 GCC 解析与堆栈

保护相关的编译选项时,如果编译选项中没有 -fstack-protector 等与 SSP 设置相关的选项,则 GCC 编译器就不会在编译程序中设置堆栈保护;否则 GCC 会调用 SSP 在函数堆栈中设置 canary,探测函数执行过程中可能发生堆栈溢出的行为;如果 GCC 解析编译选项不仅调用 SSP,而且加入了本文实现的插件,那么 GCC 会执行 plugin for SSP 编译优化遍,执行插件初始化,插件指令生成,记录表初始化,canary 更新设置这四个模块. 图 5 为插件的总体设计流程.

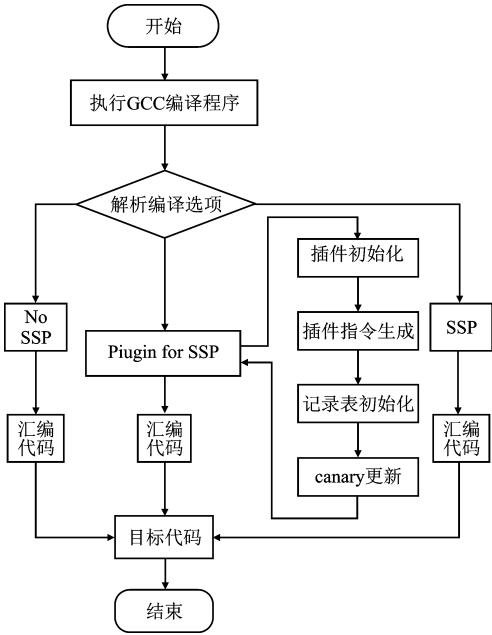


图 5 pluginforssp 设计流程图

4 实验结果与分析

本文提出的防止攻击者绕过 SSP 的堆栈溢出保护增强插件^[7]plugin for SSP 会集成到 GCC 编译工具链的编译优化过程中,作为 GCC 后端 RTL 中的一个新的编译优化过程来实现,设置这一优化遍的目的是使编译的程序调用 fork 函数时,就更新创建的子进程从父进程那里继承的堆栈中的 canary,这样目标程序中不同函数堆栈中就会设置不同的 canary.

在 GCC 命令程序中加入编译好的插件,即加入编译选项 -fplugin=. /pluginforssp. so,并且编译程序时使用 -fdump-passes 打印 GCC 当前编译过程信息. 在图 6 中看到编译过程 rtl-vartrack 后面,不仅成功的将本文实现的插件 plugin for SSP 以编译优化遍添加到 GCC 编译器中,即增加了第 263 编译优化遍,而且其后面显示 GCC 编译器开启 (ON) 了

该编译优化过程.

```
261- rtl-alignments      : ON
262- rtl-vartrack        : OFF
263- rtl-pluginforssp    : ON
264- *free_cfg           : ON
265- rtl-mach             : ON
```

图 6 编译遍 rtl-pluginforssp

要想绕过 SSP 的堆栈保护,攻击者需要知道 canary 的具体值,暴力破解 canary 值的方式有两种:一是全搜索(full search),每次猜测时用一个完整的 canary 去覆盖系统堆栈中设置的 canary 值,如果程序没有出错终止执行,则说明了猜测的 canary 值和堆栈中一样,如果程序出现堆栈溢出错误退出,则说明本次使用的 canary 值和系统堆栈中的不一样;二是将 canary 值拆分,每次只覆盖 canary 的一个字节,即基于字节的猜测(byte for byte).从表 1 中看到,在 32 位与 64 位系统中,在编译程序中使用插件 plugin for SSP 明显的增加了攻击者猜测堆栈中 canary 值的难度,因为每次子进程创建都使用新的 canary,因此 byte for byte 攻击就无法每次只探测一个字节.而 full search 攻击也只能每次都使用一个固定不变的值探测 canary,以期待此时 canary 设置的随机值与其正好一样,才能刚好猜测出当前进程的 canary,不过这种概率很小.

表 1 攻击次数

攻击方式	%100 (32bit)	Mean (32bit)	%100 (64bit)	Mean (64bit)
ssp+bfb	768	384	7×2^8	7×2^7
ssp+full	2^{24}	2^{23}	2^{56}	2^{55}
plugin+bfb	∞	∞	∞	∞
plugin+full	∞	2^{24}	∞	2^{56}

5 结束语

堆栈保护增强插件是在 SSP 的基础上实现的

防止 canary 被攻击者猜测的一种安全保护措施.在 GCC 的 SSP 中并没有提供任何保护 canary 本身安全的保护措施,本文基于 GCC 的最新插件技术实现的 plugin for SSP 是一种保护 canary 值安全有效的措施,其在很大程度上缓解了 GCC 中 SSP 防护堆栈溢出攻击的不足.

参考文献:

[1] 李文明,陈哲,李绪蓉. 缓存区溢出研究与发展[J]. 计算机应用研究, 2014, 31(9):2561-2566.

[2] Sumi A. Bounds checking for C and C++[J]. Social Science Japan Journal, 2003(1): 165-168.

[3] Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead[C]// International Conference on Software Engineering. DBLP, 2006: 162-171.

[4] 邵必林,燕佳芬,边根庆. 缓冲区溢出漏洞综合检测方法研究[J]. 微电子学与计算机, 2015, 32(7): 99-102.

[5] Wagle P, Cowan C. StackGuard: Simple stack smash protection for GCC[J]. Proc of the Gcc Developers Summit, 2003:243-255.

[6] 王曼丽. 基于 GCC 编译器插件的内核安全加固方法研究[D]. 北京 :北京交通大学, 2015.

[7] Seredinschi D A, Sterca A. Enhancing the stack smashing protection in the GCC[J]. Studia Universitatis Babes-Bolyai, Informatica, 2011(4):779-792.

作者简介:

王亚刚 男,(1972-),博士,副教授.研究方向为嵌入式系统、编译器与并行计算.

曹耀彬(通讯作者) 男,(1990-),硕士研究生.研究方向为 GCC 编译器与软件安全. E-mail:caoyaobin1990@qq.com.

王萌 女,(1994-),硕士研究生.研究方向为计算机系统结构、图像处理.