计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(8):43−52 [doi: 10.15888/j.cnki.csa.009914] [CSTR: 32024.14.csa.009914]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于软件多样性的栈溢出保护技术①

梁超毅[1], 叶子昂[1], 戴华昇[1], 张为华[1,2]

[1](复旦大学 计算机科学技术学院, 上海 200433)
[2](复旦大学 大数据研究院, 上海 200433)
通信作者: 张为华, E-mail: zhangweihua@fudan.edu.cn

摘　要: 缓冲区溢出漏洞广泛存在于由不安全的高级语言所编写的程序中. 利用缓冲区溢出漏洞, 攻击者可以实现控制流劫持等危险攻击方式. 基于 Canary 的栈保护技术是处理缓冲区溢出漏洞的一种简单有效且广泛部署的防御手段, 然而位置固定和取值相同的特点使其容易被攻击者分析和破解. 本文提出一种基于软件多样性的栈保护技术, 它以拥有随机化大小和偏移的异构 Canary 为核心, 不仅能直接抵御常规 Canary 无法处理的泄漏类和覆盖类攻击, 而且能构造出各种更加安全的多样性软件系统. 实验结果表明, 异构 Canary 在有效提升安全性的同时仅为 SPEC CPU 2017 基准程序集额外引入了不高于 2% 的编译开销和平均 3.22% 的运行开销.

关键词: 栈溢出保护; 随机化; 软件多样性; 缓冲区溢出; 控制流劫持

引用格式: 梁超毅,叶子昂,戴华昇,张为华.基于软件多样性的栈溢出保护技术.计算机系统应用,2025,34(8):43–52. http://www.c-s-a.org.cn/1003-3254/9914.html

Stack Smashing Protection Technique Based on Software Diversity

LIANG Chao-Yi[1], YE Zi-Ang[1], DAI Hua-Sheng[1], ZHANG Wei-Hua[1,2]

[1](School of Computer Science, Fudan University, Shanghai 200433, China)
[2](Institute of Big Data, Fudan University, Shanghai 200433, China)

Abstract: Buffer overflow vulnerabilities are widely present in programs written in insecure high-level languages. By exploiting buffer overflow vulnerabilities, attackers can carry out dangerous attacks such as control flow hijacking. Canary-based stack smashing protection is a simple, effective, and widely deployed defense mechanism against buffer overflow vulnerabilities. However, its characteristics of fixed locations and identical values make it susceptible to analysis and exploitation by attackers. This study proposes a stack smashing protection approach based on software diversity, which features heterogeneous Canary with randomized sizes and offsets. This approach not only directly defends against leakage and overwrite attacks that conventional Canary cannot handle, but also enables the construction of various diversified software systems with more security. Experimental results show that this technology introduces less than 2% compilation overhead and an average of 3.22% runtime overhead for the SPEC CPU 2017 benchmark set while effectively improving security.

Key words: stack smashing protection; randomization; software diversity; buffer overflow; control flow hijacking

缓冲区溢出漏洞[1]广泛存在于不安全的高级编程语言所编写的程序中. 攻击者可以利用缓冲区溢出漏洞改写进程栈上的数据, 构造出许多复杂的攻击方式. 最危险的一种攻击方式是改变函数栈帧的返回地址, 实现控制流劫持 (control flow hijacking)[2], 让程序运行到预先设计的位置, 执行恶意代码. 20 多年来, 研究者

① 基金项目: 国家重点研发计划 (2023YFB4503702)
收稿时间: 2024-12-26; 修改时间: 2025-01-26; 采用时间: 2025-02-18; csa 在线出版时间: 2025-06-24
CNKI 网络首发时间: 2025-06-25

们陆续提出了多种针对缓冲区溢出漏洞和控制流劫持的攻击方式和防御手段, 例如, 栈溢出保护 (stack smashing protection, SSP)[3–5]、地址空间随机化 (address space layout randomization, ASLR)[6]、控制流完整性 (control flow integrity, CFI)[7,8] 等. 其中, SSP 技术致力于检测缓冲区溢出这一根本问题, 是一种重要的防御手段. 常规的 SSP 技术在函数的栈帧上设置一个特定数值 (称为 Canary), 并在函数返回时检查栈帧 Canary 是否与设置时一致, 来判断函数在运行时是否发生了栈溢出. Canary 技术简单且有效, 已经在 Windows 和 Linux 等操作系统中部署 10 年以上.

然而, 常规 Canary 缺乏异构性, 即 Canary 的值保存在内存中 (称为内存 Canary), 并在任意函数被调用时原封不动地拷贝到其栈帧的某个固定位置处 (称为栈帧 Canary). 这种位置固定和取值相同的特点使得常规 Canary 存在被绕开的风险. 由于位置固定, 攻击者能轻松分析出栈帧 Canary 的位置; 由于取值相同, 任何线程的内存 Canary 或任何函数的栈帧 Canary 都代表着整个进程的 Canary. 攻击者可以利用位置固定的特点进行泄漏类攻击, 通过某些程序漏洞 (如结合溢出漏洞和 printf 函数) 将栈帧 Canary 打印出; 也可以利用取值相同的特点进行覆盖类攻击, 利用大缓冲区溢出漏洞同时覆盖内存 Canary 和栈帧 Canary 来绕开栈保护; 还可以结合两个特点进行逐字节暴力破解攻击, 在多线程系统中利用缓冲区溢出逐字节地不断尝试栈帧 Canary 的值直至将其破解, 然后应用到其他线程中.

关于 Canary 的最新研究均是针对逐字节暴力破解的攻击场景. 朱君等[9]实现了函数级别的细粒度 Canary 保护机制, 通过编译器在程序中插桩, 使进程中不同的函数使用不同的 Canary 值. 王之隆等[10]使用异或计算根据内存 Canary 简洁高效地构造栈帧 Canary, 并部署在不同函数的栈帧上. 然而, 上述防御手段仅实现了函数之间 Canary 的取值不同, 限制了 Canary 的传播以抵御逐字节暴力破解, 但栈帧 Canary 的位置依然固定, 并且依然难以抵御 Canary 覆盖类攻击.

本文提出了一种基于软件多样性的 SSP 技术. 它以异构 Canary 为核心, 为常规 Canary 添加了大小和偏移随机化变量, 将栈帧 Canary 的大小和位置变得不固定, 打破了常规 Canary 位置固定和取值相同的特点, 能够直接抵御 Canary 泄漏类和覆盖类攻击. 该技术也在异构 Canary 的基础上, 使用衍生的软件多样性技术, 构造出各种安全性更加强大的多样性软件系统.

本文通过实验验证了异构 Canary 的保护效果, 并评估了异构 Canary 引入的开销. 结果表明, 这种异构 Canary 不仅能够有效抵御 Canary 泄漏类和覆盖类攻击, 而且为程序引入的编译开销总体低于 2%, 引入的运行开销在随机化策略最复杂时也仅占 3.22%.

总结来说, 本文的主要贡献如下.

(1) 分析了常规的 SSP 技术在异构性上的缺陷;

(2) 提出了一种基于软件多样性技术的 SSP 技术, 以异构 Canary 为核心, 将栈帧 Canary 的大小和位置变得不再固定, 不仅能直接抵御常规 Canary 无法处理的泄漏类和覆盖类攻击, 而且能构造出各种安全性更强的多样性软件系统;

(3) 设计实验验证了异构 Canary 的保护效果, 并评估了其引入的开销, 结果表明其为 SPEC CPU 2017 基准程序集引入的编译开销总体低于 2%, 而不同的随机化策略的运行开销依次为 0.80%、2.08% 和 3.22%.

本文将异构 Canary 实现在 GCC 编译器中, 可以替代 Linux 操作系统中默认的 GCC 编译器. 任何 C 语言源代码 (尤其是容易遭受外部攻击的网络服务程序) 都可以部署本文提出的异构 Canary, 作为对常规 Canary 的增强. 在安全性要求较高的场景下, 也可以部署本文构造的多变体系统或动态多样性系统, 以一定的运行开销换取针对缓冲区溢出漏洞更强大的防御能力.

1 研究背景

本节首先介绍了与缓冲区溢出漏洞和控制流劫持相关的一系列攻击方法和防御手段, 然后重点分析了常规的 Canary 栈保护技术因缺乏异构性而存在的问题, 以及现有工作的不足之处. 最后介绍了软件多样性技术和 3 种主要的衍生技术.

1.1 缓冲区溢出与控制流劫持

早期的高级编程语言 (如 C 语言) 对内存的操作具有极大的自由度, 由于部分标准函数 (如 gets 函数) 缺乏对输入数据的边界检查, 攻击者可以构造一个较长的输入覆盖掉栈上的数据, 这就是缓冲区溢出漏洞.

利用缓冲区溢出漏洞可以实现控制流劫持攻击, 如图 1 所示, 攻击者通过覆盖栈上的函数返回地址使程序跳转到指定的目标. 其中, 代码复用攻击 (return oriented programming, ROP) 首先利用内存泄漏探索可复用的代码片段, 称为 gadget. gadget 通常以 return

指令结尾. 然后, 使用图灵完备的 gadget 构造一个攻击链, 再把链的地址依次填入栈中. 如此一来, 程序跳转到第 1 个 gadget 执行后, 会因为返回指令跳转到第 2 个 gadget 继续执行, 如此反复直到完成攻击意图.

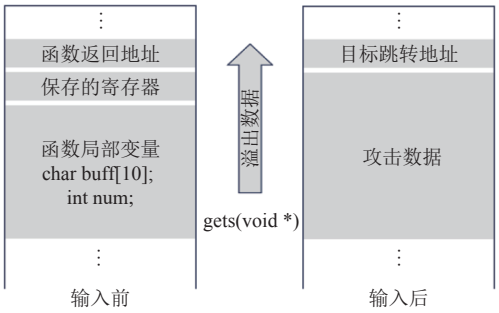

图 1 缓冲区溢出与控制流劫持

ASLR 技术旨在改变程序的内存布局, 令攻击者探明的 gadget 位置无法在程序的下一次运行中使用. 针对 ASLR, 攻击者又提出了即时代码复用攻击 (just-in-time ROP, JIT-ROP)[11], 在一次运行中探明 gadget, 构造攻击链并实施攻击. 之后, 研究者又提出了动态 ASLR 技术, 在运行时持续进行 ASLR 操作, 即持续地址空间重随机化 (continuous address space layout re-randomization)[11–13]. 这种动态技术往往为程序引入较大的运行时开销, 因此难以大规模部署.

另一种技术是 CFI, 它通过检查程序的控制流是否转移到了不合法的地址来抵御控制流劫持攻击. 细粒度的 CFI 会为程序带来大量运行时开销, 因此也很难大规模部署, 而且被证明依然存在绕过的方法[14,15].

根治控制流劫持攻击需要第一时间阻止返回地址被修改. 一种方法是重写程序, 添加边界检查, 或者使用内存安全的高级编程语言 (如 Rust). 但 C 程序往往包含海量的库, 且部分缺失源代码, 因此重写的可行性较低. 另一种方法是使用 SSP 技术, 它在栈上插入标记, 在函数返回时对比标记, 来检查返回地址是否被改写过. 这种技术简单且有效, 已经广泛部署在主流操作系统默认的 C 语言编译器和动态链接库中.

1.2 Canary 栈保护技术现状

SSP 技术大多实现为基于 Canary 的栈保护技术, 其工作原理如图 2 所示. 函数被调用时, 会将保存在某个固定内存位置的数据 (称为内存 Canary) 拷贝到栈帧上返回地址和局部变量中间的某个固定位置; 函数返回时, 比对栈帧上的 Canary (称为栈帧 Canary) 和内存 Canary, 如果不一致, 则判定发生了栈溢出.

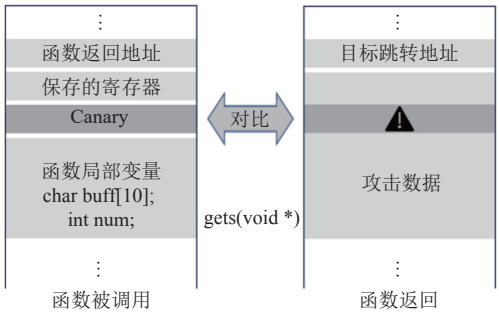

图 2 Canary 基本原理

这种方案的 Canary 具有两个特点: 一是位置固定, 即栈帧 Canary 在每个函数栈帧中的相对位置是固定的, 这让攻击者具有探明栈布局的机会; 二是取值相同, 即同一个进程的所有线程的内存 Canary 和所有函数的栈帧 Canary 的值都相同, 这让攻击者具有暴力破解或直接覆盖 Canary 的可能. 基于这些特点, 常规 Canary 技术可以被以下几种方式绕过.

(1) Canary 泄漏类攻击. 即通过某些方式获取 Canary 的值并用于修改返回地址. 尽管 Canary 的首字节默认设置为 0x00 (‘\0’) 以避免被 printf 等方式从栈上直接打印出, 但攻击者依然可以通过缓冲区溢出漏洞覆盖掉字符串截断符, 以获取 Canary. 此外, 还有其他探明栈上数据的方式, 例如利用内存泄漏等.

(2) 逐字节暴力破解. 即在频繁使用 fork 创建子进程的系统中从低到高逐字节地覆盖栈帧 Canary 的值, 如果子进程没有崩溃则固定当前字节并尝试下一个字节. 目前已有一些工作[5,16]在 fork 时更新线程局部存储 (thread local storage, TLS) 中的内存 Canary 的值, 这使得进程中如果线程不同则函数的栈帧 Canary 不同. 这种防御手段称为 RAF (renew-after-fork).

(3) Canary 覆盖类攻击. 即利用缓冲区溢出写入大量数据, 直到将栈帧 Canary 与 TLS 中的内存 Canary 一起覆盖. 这种攻击很容易应用在上述使用 RAF 防护的系统中, 因为很多线程库出于局部性考虑会把 TLS 与栈区放在临近的位置.

示例程序 1 给出了一个可以被 Canary 覆盖攻击劫持的例子, 其中第 7 行有一个较大的缓冲区溢出漏洞. 攻击者可以写入大量重复数据将栈帧 Canary 连同 TLS 数据段中的内存 Canary 一并覆盖掉. 而且在 64 位系统中, Canary 总是 8 字节对齐的, 这使得攻击者可以很方便地构造出攻击载荷, 如图 3 所示.

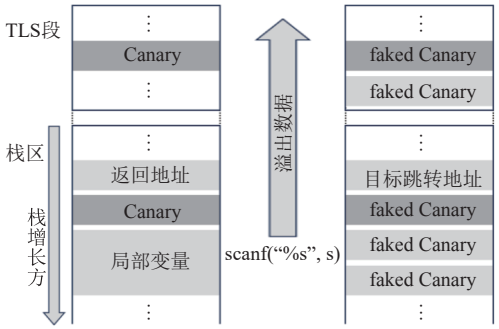

图 3 通过大量数据覆盖 Canary

示例程序 1. Canary 覆盖类攻击

```
1. void *func() {
2.     long name[8];
3.     int times;
4.     scanf("%d", &times);
5.     // 在此处写入大量数据以覆盖 TLS 数据区
6.     for(int i=0; i<times; i++) {
7.         scanf("%lx", name+i);
8.     }
9.     /* … */
10.    return 0;
11. }
12. int main() {
13.    pthread_t t;
14.    pthread_create(&t, NULL, &func, 0);
15.    /* … */
16.    return 0;
17. }
```

最新的 Canary 研究主要针对逐字节暴力破解攻击进行防御. 针对传统的 RAF 技术线程级别粒度过大的问题, DiffGuard[9]为不同函数设置不同的 Canary, 实现了更细粒度的函数级别栈保护; 即便攻击者获取了某个函数的 Canary, 也无法直接应用到其他函数中. 而 P-SSP[10]使用两个栈帧 Canary, 并通过异或运算建立栈帧 Canary 与内存 Canary 的联系, 能够便捷地构造不同的栈帧 Canary, 降低 RAF 技术引入的运行时开销.

上述研究提出的保护机制在一定程度上也能抵御 Canary 泄漏类攻击, 但依然无法抵御 Canary 覆盖类攻击. 例如, DiffGuard 中虽然每个函数的 Canary 取值都不同, 但同一个函数的内存 Canary (TLS Canary) 和栈帧 Canary 依然是相同的, 可以被同时覆盖. 而两个 0 的异或也等于 0, 因此通过构造合适的攻击载荷, 覆盖类攻击也能够绕开 P-SSP 中提出的保护机制.

1.3 软件多样性技术

软件多样性 (software diversity) 技术使用多个功能相同, 但实现或结构不同的变体, 来增加攻击者的攻击成本[17,18]. 攻击者通常依赖系统行为的可预测性来构造攻击方式, 而多样化的实现方式使得系统或组件变得不可预测, 令攻击者难以利用系统的漏洞.

软件多样性技术为软件系统带来了异构性, 它是动态异构冗余架构[19]的核心. 目前有 3 类衍生的软件多样性技术: 大规模多样性、多变体系统和动态多样性.

● 大规模多样性[20]. 将异构发挥到极致, 使用特殊的编译器为每个用户分发一份功能相同但实现上有微小差异的唯一变体, 如图 4 所示. 不同副本让攻击者难以分析并开发一个统一的攻击手段, 并且因为无法确定目标使用的具体副本, 定向攻击的难度也相应增加. 大规模多样性提高了攻击成本, 能有效防止攻击扩散, 强调整体的安全性, 但个体的安全性可能无法保证.

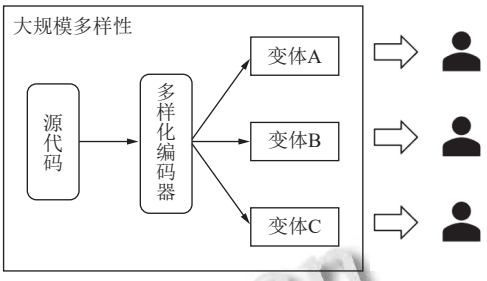

图 4 大规模多样性

● 多变体系统[19,21]. 将异构与冗余结合, 让多个功能相同但内部实现不同的变体执行同一份输入, 通过监视输出是否相同来判断系统是否遭受了攻击, 如图 5 所示. 多变体系统不需要保守任何秘密, 甚至可以抵御未知的攻击方式, 但缺点是有较大的冗余开销.

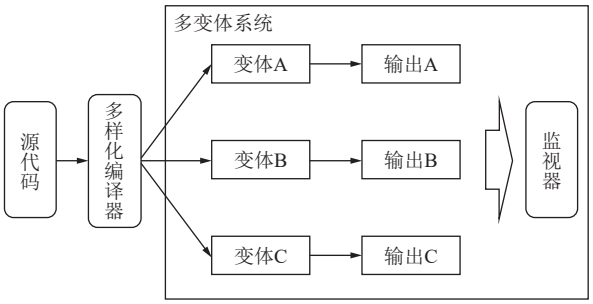

图 5 多变体系统

● 动态多样性[22–24]. 将异构与动态结合, 在程序运行时改变自身的结构或实现, 如图 6 所示. 使用这种技

术的系统被拆分为许多组件, 而每个组件具有多个不同的变体; 运行时, 这些组件来回切换变体, 从而混淆攻击者的目标, 具有很强的防御效果. 这种方法引入的总体开销比多变体系统少, 但会增加系统的延迟.

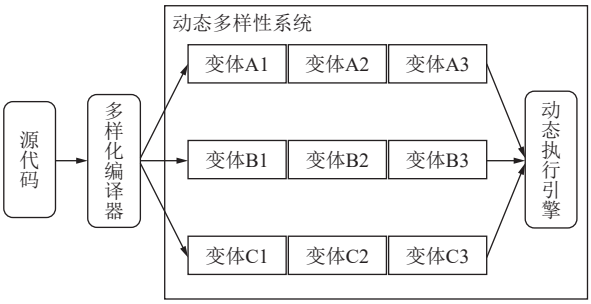

图 6 动态多样性

2 异构 Canary 设计

异构 Canary 是本文所提基于软件多样性的栈保护技术的核心. 在异构 Canary 的设计中, 为保留 Canary 技术简单且适用性强的优良特性, 需要考虑以下几点.

(1) 如何使得 Canary 的位置和取值呈现多样性?

(2) 如何权衡 Canary 的运行时开销和保护效果?

(3) 如何优化 Canary 的运行时开销和保护效果?

本节首先描述了威胁模型, 然后提出了栈内布局设计, 使用随机化变量选择要插入的 Canary 的大小和控制 Canary 插入的位置. 之后, 提出了静/动态与程序/函数级的随机化策略的 4 种组合中的 3 种, 并使用随机化池和分级取值范围优化了异构 Canary 的运行时开销和保护效果. 最后, 介绍了如何生成变体并构造出第 1.3 节提到的各种基于软件多样性的衍生系统.

2.1 威胁模型

本文考虑基于缓冲区溢出且常规 Canary 泄漏导致的控制流劫持攻击, 做出如下假设.

(1) 被攻击程序无缓冲区边界检查等机制, 攻击者能使用缓冲区溢出漏洞覆盖进程堆栈上的数据.

(2) 攻击者已知晓进程堆栈在一般情况下的布局, 包括常规 Canary 在堆栈上摆放的位置.

(3) 攻击者能破解或覆盖常规 Canary 的值.

这些假设建立在常规 Canary 位置固定和取值相同的特性上, 实现方式在第 1.2 节已经介绍, 不再赘述.

2.2 栈内布局

本文从大小和偏移这两个维度对 Canary 的异构性进行补足, 使其位置和取值呈现多样性. 异构 Canary 的大小由随机化变量 size 来决定, 位置根据随机化变量 offset 调整, 如图 7 所示.

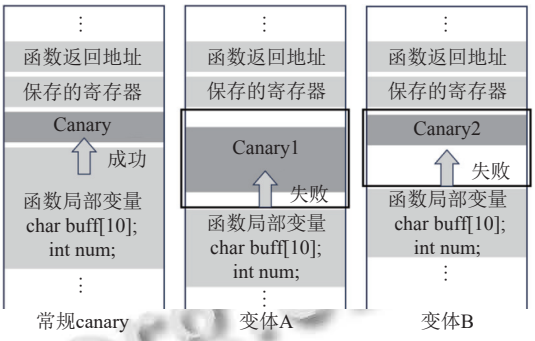

图 7 随机化大小和偏移的异构 Canary

● 随机化变量 size. 不同于常规的 32 位或 64 位定长 Canary, 异构 Canary 大小并不固定, 这打破了 Canary 在栈上大小的认知, 也间接改变了 Canary 在栈上的布局. 这种设计需要在每个被保护函数的栈帧上预留一块随机大小的空间 (称为 padding, 如图 7 中矩形框所示), 以容纳大小不一的异构 Canary.

● 随机化变量 offset. 如果异构 Canary 与 padding 的相对位置依然是确定的, 攻击者仍然有泄漏或覆盖 Canary 的攻击面. 故在大小之外, 异构 Canary 还引入了偏移, Canary 实际插入的位置由 padding 的首地址和偏移共同决定.

实现上, 这种异构 Canary 的插入和比对由编译器从源代码生成目标代码时完成. 函数 prologue 的常规操作完成后, 会根据随机化 size 获取合适的 Canary, 再根据 padding 的首地址和随机化 offset 计算 Canary 的位置并插入, 伪代码如算法 1 所示. 其中, padding 的首地址即 padding 的最高位地址减去 padding 的大小 (对应图 7 中矩形框的下边线).

算法 1. 函数 prologue (部分)

```
1. unsigned int padding[PADDING_SIZE];
2. dest=ADDR(padding)+RANDOM_OFFSET;
3. switch(RANDOM_SIZE) {
4. case 32bit:
5.     movl CANARY_32, (dest)
6. case 64bit:
7.     movq CANARY_64, (dest)
8. case 128bit:
9.     movq CANARY_64, (dest)
10.    movq CANARY_64+8, (dest+8)
11.    …
12. }
```

需要强调的是, padding 的大小虽然也是随机的, 但它与 size 和 offset 的性质并不相同, 因为 padding 需要在编译时预留且其大小总是在编译时确定的 (而 size 和 offset 可以不在编译时而是在装载时确定). 因此, 若无特殊说明, 本文的“随机化变量”仅指代变量 size 和变量 offset.

同样地, 函数 epilogue 的常规操作完成后, 根据变量 size、padding 的首地址和变量 offset 判断 Canary 的位置并与内存中的值比对, 伪代码如算法 2 所示.

算法 2. 函数 epilogue (部分)

```
1. dest=ADDR(padding)+RANDOM_OFFSET;
2. switch(RANDOM_SIZE) {
3. case 32bit:
4.     cmpl CANARY_32, (dest)
5. case 64bit:
6.     cmpq CANARY_64, (dest)
7. case 128bit:
8.     cmpq CANARY_64, (dest)
9.     cmpq CANARY_64+8, (dest+8)
10.    …
11. }
12. jne FAILED
```

本文并不修改 Canary 本身的值的生成方式, 而是复用编译器生成的常规 Canary, 如上述算法 2 中 32 位的 CANARY_32 和 64 位的 CANARY_64. 因此, 本文将随机化变量 size 的取值限制为 32 位、64 位和 128 位, 并使编译时预留的 padding 大小不小于最大 size (记为 max_size) 的两倍, 种类不超过 16 种, 故 padding 的取值范围是[max_size/4, (max_size/4)+15] 字节. 上述 size 和 padding 的取值范围使得 offset 的合法取值范围是 [0, max_size/8] 字节. 此外, size 和 offset 有多种可选的随机化时机和粒度, 即, 其值在何时设置, 以及其值是全局唯一的还是每个函数独有的.

2.3 随机化策略

随机化的时机可以分为静态和动态, 前者在编译时确定随机化变量的值, 但每次编译的取值各不相同; 后者在运行时确定随机化变量的值, 随机化变量被编译器创建为外部声明, 目标代码中它们不再是立即数, 而是对外部变量的引用, 由动态链接库提供实际的值.

随机化的粒度可以分为程序级和函数级, 前者为程序中所有函数生成全局唯一的一组随机化变量, 后者在编译每个函数时单独为其提供一组随机化变量.

在随机化时机和粒度的 4 种组合方式中, 静态-程序级随机化与常规 Canary 的本质差别仅是让 size 变得更大了, 因此本文提出以下 3 种随机化策略.

● 静态-函数级. 编译时为每个函数固定一组随机化变量. 这种策略下程序中每个函数的栈帧都有不同的 Canary 栈内布局, 但多次运行之间栈内布局不会改变. 尽管系统具备了丰富的异构性, 但攻击者依然能利用程序的重启机制不断尝试以暴力破解 Canary 的布局.

● 动态-程序级. 运行时为整个程序选择一组随机化变量. 这种策略下程序的每次运行之间都会有不同的 Canary 栈内布局, 但运行时每个函数栈内布局都相同. 尽管一个函数泄漏的 Canary 可被用于攻击其他函数, 但攻击者不能利用程序上一次运行时获取的信息, 这无疑对攻击的方式和漏洞的特点有更严苛的要求.

● 动态-函数级. 运行时为每个函数选择一组随机化变量. 这种策略下程序的每个函数和每次运行之间的 Canary 在栈内的布局都不相同, 这使得系统中 Canary 的布局极富多样性: 无论是程序的上一次运行还是程序中的其他函数都无法给出用于攻破目标函数的提示. 但这种策略的缺点在于, 每次函数调用时都需要进行一次随机化变量的获取, 不仅会消耗 CPU 资源, 而且随机化变量本身的内存占用也不容忽视.

需注意的是, 上述随机化变量仅包含 size 和 offset. 尽管 padding 的大小也是随机的, 但 padding 需要在编译时预留且运行时不可变, 因此固定为静态-函数级.

不同场景需要的随机化策略不同. 其中, 动态-函数级随机化提供了最高的灵活度, 但变量生成和管理的开销也最高. 如果不精心设计和优化, 引入的运行时开销将会变得不可接受.

2.4 随机化池

如果在动态-函数级策略的每次函数调用时即时生成随机化变量, 将会带来如下问题.

(1) 延缓运行性能. 每次函数调用时都要生成 size 和 offset 随机数, 这会引入很多 CPU cycle 开销.

(2) 提高内存占用. 为了保存两个随机化变量, 每个函数的内存占用都要增加 8 字节.

本文采用随机化池来优化这种策略的性能: 在系统中预设一定数量的随机化变量池, 每个池中保存了若干组随机化变量 size 和 offset. 编译时, 编译器为每个函数分配一组随机的下标, 以引用随机化变量池中的一组变量; 运行时, 动态链接器为程序随机链接一个

随机化变量池, 因此即便下标确定, 每次运行指向的随机化变量值也不同, 如图 8 所示.

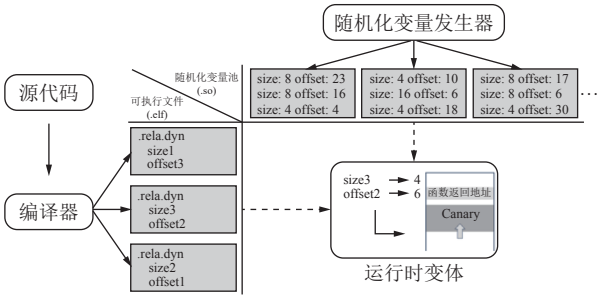

图 8 随机化池

这种方案以随机性的略微损耗为代价, 使得每个函数获取随机化变量的操作缩减为对内存的两次访问, 同时也降低了维护随机化变量的内存开销, 因为函数与池中变量可以是多对少的映射关系. 此外, 这种方案将随机化变量的获取分散到编译和运行两个阶段, 编译时函数对池中变量的引用是随机的, 运行时程序使用的池也是随机的, 这为系统带来了额外的异构性.

静态-函数级策略和动态-程序级策略都可以看作是动态-函数级策略的特例, 因此也可以使用上述随机化池. 静态策略下, 编译器利用随机化池生成立即数; 程序级策略下, 所有函数的随机下标都指向随机化池中的第 1 组随机化变量. 此外, 由于 padding 的大小须在编译时生成和确定, 本文不将其加入随机化池.

2.5 分级取值范围

随机化池的一个简单策略是每个池中都有每种取值范围的 size 和 offset. 然而, 程序中不同性质的函数对保护程度的要求不同. 例如, 有输入缓冲区的函数显然比没有缓冲区的函数需要更强的保护. 反之, 为没有缓冲区的函数使用与有输入缓冲区的函数相同取值范围的随机化变量是不必要且浪费资源的.

本文为随机化池设置了分级取值范围, 如图 9 所示. 这种划分方式也和 GCC 编译器使用的策略一致. 最需要保护的函数划分到 DEFAULT 范围, 这类函数通常具有较大的缓冲区, 或使用了动态栈分配, 对应的随机化池为 High 池, size 可取 64 位或 128 位. 其他范围以此类推. 可以通过编译器选项调整程序使用的保护程度, 例如, 使用 STRONG 级则只为 STRONG 范围内的函数做 Canary 保护, 其中也包括 DEFAULT 范围内的函数 (真子集); 使用 ALL 级则保护所有函数.

分级取值范围仅在函数级随机化中使用. 程序级随机化由于所有函数共用一组随机化变量, 因此不存在上述分级, size 的取值范围固定为 {32, 64, 128}.

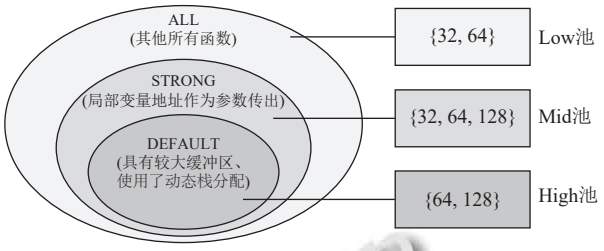

图 9 分级取值范围

2.6 多样性系统构造

本节提出的基于软件多样性的栈保护技术可用于构造上文提到的 3 类衍生的多样性系统. 其中, 大规模多样性 (如图 1) 和多变体系统 (如图 2) 的构造依赖于程序的大量变体, 而本文的异构 Canary 技术可以很方便地满足这个要求. 使用 3 种随机化策略的任何一种都可以生成充足的变体.

构造动态多样性系统 (如图 3) 则更加复杂. 本文以 MELF[24] 系统为例来说明构造的过程. MELF 是一种动态多样性技术, 它将同一个函数的多个变体编译到同一个虚拟地址, 确保运行时切换函数变体后所有对该函数的引用都能保持正确. 使用本文提出的异构 Canary 技术构造 MELF 系统需要动态-函数级随机化, 其大致过程如图 10 所示.

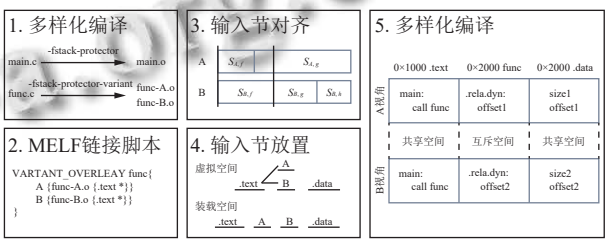

图 10 利用异构 Canary 构造动态多样性系统

编译时, 实现了异构 Canary 技术的 GCC 编译器为每个源代码文件生成多个目标文件, 每个目标文件的函数使用不同的随机化变量; 装载时, 动态链接一个随机化池, 结合 MELF 系统的执行框架, 即可在运行时动态地切换程序中函数的变体, 实现动态随机化.

3 实验分析

本节设计了模拟攻击实验和性能实验, 旨在验证异构 Canary 对泄漏类和覆盖类攻击的抵御能力, 并评

估异构 Canary 为程序的编译和运行引入的性能开销. 此外, 本节还讨论了基于异构 Canary 构造的多样性系统的安全性, 并结合性能实验的数据, 比较分析了不同随机化策略下异构 Canary 为程序带来的开销.

3.1 实验环境

本文的异构 Canary 技术在 GCC 11.4.0 版本中实现, 实验的环境如表 1 所示.

表 1 实验环境

实验环境	配置信息
CPU型号	Intel® Xeon® CPU E7-4807 @ 1.87 GHz
内存信息	256 GB DDR4
OS内核	Linux 5.4.0-148-generic
OS版本	Ubuntu 22.04.2 LTS
测试集程序	SPEC CPU 2017
编译器选项	-static -O3 -no-pie

3.2 安全性分析

评估异构 Canary 是否能够抵御泄漏类和覆盖类攻击的方法是, 使用此类攻击现有的攻击载荷或使用相同的思路构造新的攻击载荷, 并对异构 Canary 实施攻击, 观察攻击是否依然能够生效.

本文设计了两组模拟攻击实验验证异构 Canary 对泄漏类和覆盖类攻击的抵御能力. 对于 Canary 泄漏攻击, 由于 Canary 泄漏的方式多种多样, 本文假设攻击者已经知晓了内存 Canary 的值. 因此, 第 1 组实验通过一个已知的 Canary 来构造攻击载荷, 试图覆盖栈帧上返回地址附近的 Canary. 而第 2 组实验针对第 1.2 节中的示例程序 1, 通过一个 8 字节重复的攻击载荷对其进行攻击. 实验结果表明, 上述两种攻击都未能成功.

对于 Canary 泄漏攻击, 由于异构 Canary 的位置并不固定, 上述第 1 组实验中基于常规 Canary 在栈帧上的布局构造的攻击载荷自然无法成功. 即便重新构造攻击载荷, 由于攻击者无法事先得知异构 Canary 的大小, 也难以推测出异构 Canary 在进程堆栈上的位置, 因此只能盲目尝试其在栈帧上的位置, 因此难以成功.

对于 Canary 覆盖攻击, 由于异构 Canary 在栈帧上的位置并不一定与内存 Canary 对齐, 而且非法地覆盖了变量 size 也会导致错误, 因此第 2 组实验的攻击也未能成功. 此外, 由于异构 Canary 打乱了函数栈帧的布局, 攻击者无法精确地定位栈帧上的返回地址, 这使得覆盖类攻击更加难以实现控制流劫持.

异构 Canary 也能根据第 2.6 节介绍的方法构造各种多样性系统, 这些多样性系统有很强的安全性. 考虑异构 Canary 的多样性, 其 size 的取值有 3 种, offset 的取值有 17 种, padding 的大小取值有 16 种, 因此单个函数栈帧上 Canary 的布局就有 816 种. 假设攻击者能用某种方式知晓函数某次运行的 Canary 的值 (这本身就相当困难), 在大规模多样性系统中, 攻击者构造的一个攻击载荷成功的概率仅有 1/816. 在多变体系统中, 一次攻击同时攻破两个冗余执行体的概率也仅有 $(1/816)^2=1/665856$, 而冗余执行体的数量还能增加. 在动态多样性系统中, 攻击者对某个组件的一次攻击成功的概率仅为 1/816, 且攻击者只有有限次重试的机会, 因为组件会发生改变. 需要强调的是, 在结合 RAF 技术的情况下, 攻击者无法利用系统的线程创建来重试, 而其他场景下也很难找到重试的机会.

3.3 性能测试

性能实验评估了异构 Canary 技术的编译时和运行时性能开销. 编译实验使用 ALL 级保护范围, 以反映编译开销的上界. 开启异构 Canary 为 SPEC CPU 2017 的部分测试程序带来的额外编译时间占原时间的比例如表 2 所示, 其中 S/D 表示静态/动态, P/F 表示程序级/函数级. 可以看到, 即便是 ALL 级, 异构 Canary 引入的编译开销也极低, 最坏情况下不超过 2%.

表 2 ALL 级编译性能损耗 (%)

测试程序	S-F	D-P	D-F
perlbench	0.21	0.11	0.23
gcc	1.76	1.40	1.87
mcf	0.54	0.23	0.50
lbm	0.25	0.10	0.33
xalancbmk	0.38	0.17	0.19
deepsjeng	1.25	1.08	1.69
imagick	0.31	0.01	0.22
leela	0.19	0.06	0.16
nab	0.21	0.13	0.33
xz	0.20	0.15	0.22
Average	**0.53**	**0.344**	**0.574**

动态-函数级随机化策略引入的编译开销最高, 平均约为 0.57%. 静态-函数级随机化会比动态-程序级随机化带来更高的编译开销, 这是因为前者在编译时需要生成 3 组随机数, 分别是 padding 的大小, 变量 size 和变量 offset 的值; 而后者只需生成 padding 的大小, 因为变量 size 和变量 offset 固定指向随机化池中的第 1 组变量, 而随机化池也仅需包含 1 组变量.

运行实验则使用 STRONG 级保护范围, 因为实际场景下为所有函数都添加 Canary 保护是不必要的. 开启异构 Canary 为 SPEC CPU 2017 的部分测试程序带来的额外运行时间占原时间的比例如表 3 所示.

表 3 STRONG 级运行性能损耗 (%)

测试程序	S-F	D-P	D-F
perlbench	0.53	1.40	2.54
gcc	0.87	2.67	3.15
mcf	0.88	1.80	3.67
lbm	0.93	1.27	2.60
xalancbmk	0.99	3.20	3.86
deepsjeng	0.47	2.33	3.11
imagick	0.62	1.50	2.48
leela	1.20	2.45	3.20
nab	0.90	1.26	3.72
xz	0.64	2.90	3.90
Average	**0.80**	**2.08**	**3.22**

动态-函数级随机化策略引入的运行开销最高, 平均约为 3.22%. 动态-程序级随机化会比静态-函数级随机化带来更高的运行开销, 这是因为前者在运行时使用的随机化变量 size 和 offset 在编译时已经变成了目标代码中的立即数, 而后者需要通过下标引用的方式获取动态链接库中的随机化变量, 因此每次读取随机化变量都要使用一条访存指令.

需要指出的是, 本文在实现中将 Canary 的值与随机化变量 size 和 offset 的值保存在相邻的内存位置, 充分利用了缓存性能, 这使得读取 Canary、size 和 offset 这 3 个值的操作与原本只读取 Canary 这一个值的操作的访存开销相差不大, 也是为什么动态-函数级随机化引入的运行时开销依然较低的原因之一. 这也增加了覆盖类攻击在异构 Canary 上成立的难度, 因为 size 可能会被攻击载荷非法覆盖, 导致函数返回时异常退出.

4 相关工作

基于 Canary 的栈溢出保护技术提出已久, 但对 Canary 增强的工作几乎都围绕多线程场景的逐字节暴力破解展开. 在 Marco-Gisbert 等[5]提出 RAF 之后, Petsios 等[16]通过维护栈 Canary 地址链表解决了 TLS 与已有栈帧的 Canary 不一致的问题. DiffGuard[11]在不同的函数中使用不同的 Canary, 限制了 Canary 的传播. P-SSP[12]在栈上部署了两个 Canary 使其异或与 TLS Canary 相等, 进一步优化了 RAF 策略, 也提出了许多扩展增强保护能力. Hawkins 等[25]实现了 Canary 的动态随机化, 能在运行时改变 TLS 和栈帧 Canary 的值, 虽然很好地扩展了 Canary 的动态安全性, 但会引入约 24% 的运行时开销.

本文提出的基于软件多样性的栈保护技术并不针对多线程程序, 而是面向通用场景. 与其他工作致力于将 Canary 的取值多样化不同, 本文将栈帧 Canary 的大小和位置随机化, 使其在栈上的布局变得不再固定. 这增强了 Canary 的异构性, 同时也为各种软件多样性系统的构造提供了先决条件.

5 总结与展望

本文提出了一种基于软件多样性的栈保护技术, 为 Canary 引入了随机化的大小和偏移, 补充其在栈上布局的异构性, 使其能够抵御常规 Canary 无法处理的泄漏类和覆盖类攻击. 本文设计了 3 种随机化策略以适应不同场景的要求, 通过随机化池和分级取值优化了其性能开销. 这种异构 Canary 增强了系统的安全性, 并且可用于构造各种更安全的多样化软件系统. 实验表明异构 Canary 引入的编译开销极低, 且引入的运行开销在使用最复杂的随机化策略时也仅有 3.22%.

未来的研究拟将本文提出的技术扩展到二进制级, 直接对现有的二进制程序进行 Canary 异构性的增强, 不再需要对源代码重编译, 以进一步提升其可用性.

参考文献

1 Lhee KS, Chapin SJ. Buffer overflow and format string overflow vulnerabilities. Software: Practice and Experience, 2003, 33(5): 423–460. [doi: 10.1002/spe.515]

2 Butt MA, Ajmal Z, Khan ZI, *et al*. An in-depth survey of bypassing buffer overflow mitigation techniques. Applied Sciences, 2022, 12(13): 6702. [doi: 10.3390/app12136702]

3 Cowan C, Pu C, Maier D, *et al*. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proceedings of the 7th USENIX Security Symposium. San Antonio: USENIX Association, 1998. 5.

4 Tan X, Mohan S, Armanuzzaman M, *et al*. Is the Canary dead? On the effectiveness of stack canaries on microcontroller systems. Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. Avila: ACM, 2024. 1350–1357.

5 Marco-Gisbert H, Ripoll I. Preventing brute force attacks against stack Canary protection on networking servers. Proceedings of the 12th IEEE International Symposium on

Network Computing and Applications. Cambridge: IEEE, 2013. 243–250.
6 Shacham H, Page M, Pfaff B, *et al*. On the effectiveness of address-space randomization. Proceedings of the 11th ACM Conference on Computer and Communications Security. Washington: ACM, 2004. 298–307.
7 Li JK, Wang Z, Bletsch T, *et al*. Comprehensive and efficient protection of kernel control data. IEEE Transactions on Information Forensics and Security, 2011, 6(4): 1404–1417. [doi: 10.1109/TIFS.2011.2159712]
8 Burow N, Carr SA, Nash J, *et al*. Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR), 2017, 50(1): 16.
9 朱君. 基于 Canary 的增强型栈保护技术研究 [硕士学位论文]. 南京: 南京大学, 2017.
10 王之隆. 基于多态 Canary 的栈保护技术研究 [硕士学位论文]. 南京: 南京大学, 2019.
11 Snow KZ, Monrose F, Davi L, *et al*. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. Proceedings of the 2013 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2013. 574–588.
12 Williams-King D, Gobieski G, Williams-King K, *et al*. Shuffler: Fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 367–382.
13 Nikolaev R, Nadeem H, Stone C, *et al*. Adelie: Continuous address space layout re-randomization for Linux drivers. Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 483–498.
14 Göktas E, Athanasopoulos E, Bos H, *et al*. Out of control: Overcoming control-flow integrity. Proceedings of the 2014 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2014. 575–589.
15 Xu JH, Di Bartolomeo L, Toffalini F, *et al*. Warpattack: Bypassing CFI through compiler-introduced double-fetches. Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2023. 1271–1288.
16 Petsios T, Kemerlis VP, Polychronakis M, *et al*. Dynaguard: Armoring Canary-based protections against brute-force attacks. Proceedings of the 31st Annual Computer Security Applications Conference. Los Angeles: ACM, 2015. 351–360.
17 Larsen P, Homescu A, Brunthaler S, *et al*. SoK: Automated software diversity. Proceedings of the 2014 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2014. 276–291.
18 Jajodia S, Ghosh AK, Swarup V, *et al*. Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats. New York: Springer, 2011: 77–98.
19 邬江兴. 网络空间拟态防御研究. 信息安全学报, 2016, 1(4): 1–10.
20 Franz M. E unibus pluram: Massive-scale software diversity as a defense mechanism. Proceedings of the 2010 New Security Paradigms Workshop. Concord: ACM, 2010. 7–16.
21 Cox B, Evans D, Filipi A, *et al*. N-variant systems: A secretless framework for security through diversity. USENIX Security Symposium. Vancouver: USENIX Association, 2006. 9.
22 Priamo G, D'Elia DC, Querzoni L. Principled composition of function variants for dynamic software diversity and program protection. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester: ACM, 2023. 183.
23 Crane S, Homescu A, Brunthaler S, *et al*. Thwarting cache side-channel attacks through dynamic software diversity. Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015). San Diego: Internet Society, 2015: 1–14.
24 Töllner D, Dietrich C, Ostapyshyn I, *et al*. MELF: Multivariant executables for a heterogeneous world. Proceedings of the 2023 USENIX Annual Technical Conference. Boston: USENIX, 2023. 257–273.
25 Hawkins WH, Hiser JD, Davidson JW. Dynamic Canary randomization for improved software security. Proceedings of the 11th Annual Cyber and Information Security Research Conference. Oak Ridge: ACM, 2016. 9.

(校对责编: 王欣欣)