SoK: Eternal War in Memory

IEEE Symposium on Security and Privacy 2013

INTRODUCTION

- 建立了一个内存错误攻击的模型，并基于该模型评测了不同的可施行的安全措施
- 分类
- 评估比较提出的修复漏洞方法的性能，兼容性，稳健性
- 讨论了为什么有些提出方法没有进行使用的原因，以及对于一个修复方法的必需标准

ATTACKS

Memory corruption 内存崩坏

spatial error: 使用出界指针

temporal error: 使用野指针

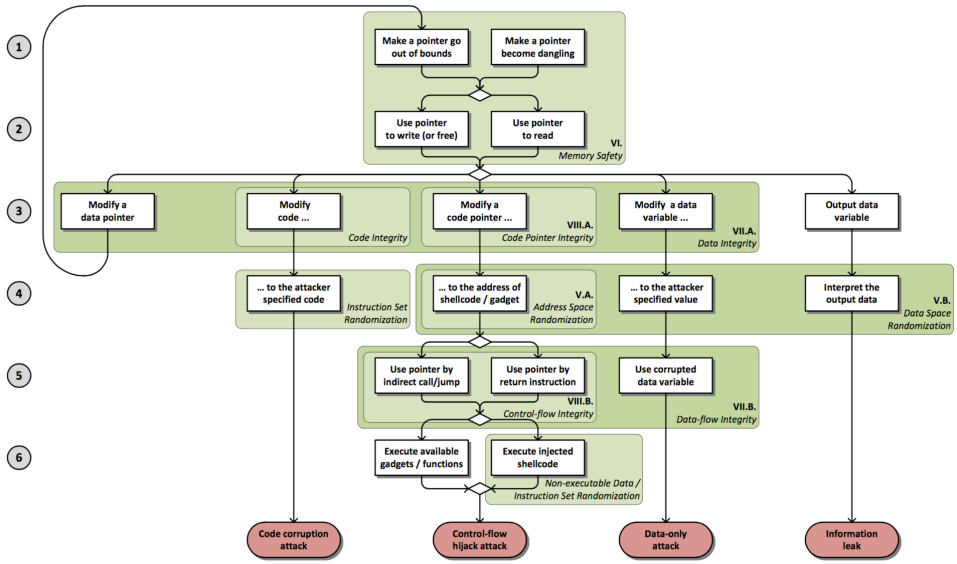

Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Code corruption attack

解决方案

Code Integrity: 所有内存页上的代码为只读形式

不足：Code Integrity 不支持self- modifying code 和 Just-In-Time (JIT) compilation， 但是现今每个主流浏览器都包含了编译Javascript的JIT编译器或者Flash。

Control-flow hijack attack 程序流劫持

Code Pointer Integrity

Address space randomization

Data-flow integrity

Non-execute Data : ROP JOP

Non-executable Data 和 Code Integrity 方法结合产生了 W⊕X (Write XOR Execute) policy, 一个内存页可以被写或可执行，但不能同时既可以写又可执行 （JIT compilation or self-modifying code 不适用）

Instruction Set Randomization

简略提到了

eliminate useful code chunks (for ROP) from the code by the compiler [20], or by binary rewriting [21].

higher-level policies which only confine the attacker's access, including per- missions, mandatory access control or sandboxing policies enforced by SFI [22], XFI [23] or Native Client

Data-only attack

Data Integrity: 包含 Code Integrity 和 Code Pointer Integrity

Data Space Randomization

Data-flow integrity

Information leak

full Data Space Randomization

CURRENTLY USED PROTECTIONS AND REAL WORLD EXPLOITS

常用保护机制: stack smashing protection、DEP/W⊕X、ASLR. Windows 提供了一些特殊的保护机制，例如保护 heap metadata 的 SafeSEH 和 exception handlers (SEHOP).

- stack smashing protection: cookie
- SafeSEH、SEHOP: 在使用之前检查异常处理指针

  缺点：只检查了一些特殊指针，例如返回地址和异常处理指针。检查可以被bypass。（例如，cookie不能检查覆盖，indexing error）
- DEP/W⊕X: 最基本攻击措施：信息泄露

APPROACHES AND EVALUATION CRITERIA

防护措施分为两类：

- probabilistic protection
- deterministic protection

Probabilistic solutions

Instruction Set Randomization, Address Space Randomization, or Data Space Randomization, build on randomization or encryption

deterministic protection

实现了低层次访问监视器，当监视到程序执行违背安全策略时，立刻停止运行

传统的访问监视器采用高层次的策略，例如文件系统权限，并且在内核中实现（例如系统调用）

低层次访问监视器（例如 Memory Safety or Control-flow Integrity）可以通过两种方式实现：硬件和代码嵌入

W⊕X采用硬件实现

Dynamic (binary) instrumention(例如 Valgrind, PIN, DynamoRIO, or libdetox)动态插入安全检查，但降低了运行速度。

low overload: shadow stack

high overload: taint checking、ROP detectors

评判标准

- Protection

  改进策略：精确性由误报率和漏报率决定

- Cost

  Perfomance overhead

  Memory overhead

- Compatibility

  source compatibility：不需要修改源代码

  Binary compatibility：依旧能和未修改的库进行链接

  Modularity support：能独立地支持适用于每个二进制文件

PROBABILISTIC METHODS

依赖于随机化和密钥，有三种主要的方法

Address Space Randomization

最突出的：ASLR

在大多数的Linux发行版本中，只有库函数代码位置被随机化，主模块代码位置没有被随机。大多数程序没有实现PIE(Position Independent Executables)。

32位机器可以使用爆破或者消随机化攻击。消随机化攻击（de-randomizayion attacks）将payload重复放在内存中。

另一种方法是 partial pointer overwrites。改写指针的最后一个字节，使之指向相近的地址。

信息泄露是最有效的攻击方法。

尽管Self-Transforming Instruction Relo- cation (STIR)等手段能在开始阶段用改变基本块顺序等手法阻扰rop，但他们并不能阻止return-to-libc攻击。

指针加密：PointGuard，加密在内存中的所有指针，并当指针加载到寄存器中才解密指针。使用同一个key进行xor加密，key容易被回复，同时不是binary source compatible.

Data Space Randomization

随机化了内存中数据的表达形式，而不是位置。加密了所有的变量，使用不同的key。当从内存载入载出时加解密数据。

不仅能防止控制流劫持，还能防止data-only exploits。

开销 15%，not binary compatible and modularity support。

MEMORY SAFETY

Spatial safety with pointer bounds

Pointer based bounds checking: CCured、Cyclone 将指针拓展为结构，添加信息，不符合binary compatibility

SoftBound:将metadata与指针分隔，使用哈希表或shadow memory space映射指针到metadata。 但 performance overhead高，达到67%，对不受保护的库文件不兼容，同步不及时。

Spatial safety with object bounds

将信息与object相连，而不是指针

能及时同步，但当指针没有被使用时，不管指针是否越界。可能出现漏报，在对象或结构中的内存溢出不能被发现，因为可以通过memset清零等。

Baggy Bounds Checking (BBC)，trades memory for performance and adds padding to every object so that its size will be a power of two and aligns their base addresses to be the multiple of their (padded) size

BBC速度较快

Temporal safety

问题：use-after-free、double-free

Special allocators

Cling:替代malloc，只允许相同类型，alignment的对象进行地址空间重利用

Object based approaches

Valgrind's Memcheck tool and AddressSanitizer

在shadow memory space中标记释放了的地址位置

Pointer based approaches

将界限、分配信息与指针联系

CETS 在全局字典中存储代表分配信息的validity bit，与SoftBound一起使用，可以达到Memory Safety,但存在binary compatibility问题

GENERIC ATTACK DEFENSES

比Memory Safety稍弱的策略：Data Integrity and Data-flow Integrity，防止控制流劫持和non-control data attacks，当不防止信息泄露。

Data Integrity

不保护temporal safety，只防护不正确的内存写，不保护内存读

Integrity of "safe" objects

static pointer analysis： identifies the unsafe pointers together with their points-to sets, i.e., their potential target objects（unsafe obejects）

在shadow memory区域插入代码，在unsafe object创建时进行标记，析构时取消标记。每一次写时检查其位置是否有在shadow memory标记。

binary compatibility、modularity support

Integrity of points-to sets

WIT(Write Integrity Testing),restricting each pointer dereference to write only objects in its own points-to set

WIT uses points-to analysis at compile time tocompute the control-flow graph and the set of objects that can be written by each instruction in the program. Then it generates code instrumented to prevent instructions from modifying objects that are not in the set computed by the static analysis, and to ensure that indirect control transfers are allowed by the control-flow graph.

WIT 不保护读。

not binary compatibility、modularity support

Data-flow Integrity

a security policy that dictates every runtime data flow of a program execution should be a true dataow of that program

DFI 在虽有数据使用前检查读操作.

DFI restricts reads based on the last instruction that wrote the read location.

not binary compatible

CONTROL-FLOW HIJACK DEFENSES

Code Pointer Integrity

不变的代码指针：在只读内存页中

Control-flow Integrity

Dynamic return integrity

- stack cookies or canaries：不防护 indirect calls and jumps，可通过 direct overwrite和信息泄露绕过。因代价低和无兼容性问题被广泛使用。
- shadow stacks RAD 提出guard pages or switching write permission to protect the shadow stack area.

Static control-flow graph integrity

statically determining the valid targets of not only calls, but also function returns

---

DISCUSSION

	Policy type (main approach)	Technique	Perf. % (avg/max)	Dep.	Compatibility	Primary attack vectors
Generic prot.	Memory Safety	SofBound + CETS	116 / 300	×	Binary	—
		SoftBound	67 / 150	×	Binary	UAF
		Baggy Bounds Checking	60 / 127	×	—	UAF, sub-obj
	Data Integrity	WIT	10 / 25	×	Binary/Modularity	UAF, sub-obj, read corruption
	Data Space Randomization	DSR	15 / 30	×	Binary/Modularity	Information leak
	Data-flow Integrity	DFI	104 / 155	×	Binary/Modularity	Approximation
CF-Hijack prot.	Code Integrity	Page permissions (R)	0 / 0	✓	JIT compilation	Code reuse or code injection
	Non-executable Data	Page permissions (X)	0 / 0	✓	JIT compilation	Code reuse
	Address Space Randomization	ASLR	0 / 0	✓	Relocatable code	Information leak
		ASLR (PIE on 32 bit)	10 / 26	×	Relocatable code	Information leak
	Control-flow Integrity	Stack cookies	0 / 5	✓	—	Direct overwrite
		Shadow stack	5 / 12	×	Exceptions	Corrupt function pointer
		WIT	10 / 25	×	Binary/Modularity	Approximation
		Abadi CFI	16 / 45	×	Binary/Modularity	Weak return policy
		Abadi CFI (w/ shadow stack)	21 / 56	×	Binary/Modularity	Approximation

Table II
THIS TABLE GROUPS THE DIFFERENT PROTECTION TECHNIQUES ACCORDING TO THEIR POLICY AND COMPARES THE PERFORMANCE IMPACT, DEPLOYMENT STATUS (DEP.), COMPATIBILITY ISSUES, AND MAIN ATTACK VECTORS THAT CIRCUMVENT THE PROTECTION.

only solutions with negligible overhead are adopted in practice