

2023-2024学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第13次课] 非控制数据攻击及防御

授课教师：陈李维

授课时间：2024. 5. 20

内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

内容概要

- **非控制数据攻击**
 - CFB
 - DOP
- **对非控制数据攻击的防御**
 - DSR
 - DFI
- **总结**

- 内存漏洞广泛存在于计算机的各种软件中，所以，攻击者能够利用内存漏洞修改或者读取内存中的数据。
- 因此，我们需要进一步研究，在攻击者能够**任意修改或读取内存数据**的前提下，攻击者到底是如何进行**攻击**的，我们又应该如何针对这些攻击行为进行**防御**？

○假设攻击者能够任意修改或读取内存**数据**，那么想要控制系统的**运行**，一共有哪些攻击方法？

○内存数据包括哪些类型？

- 指令（代码）

- 数据

 - 地址数据

 - 指令地址（代码指针，返回地址等）

 - 数据地址（数据指针）

 - 正常数据

 - 关键数据

 - 非关键数据

栈(Stack)	内存高地址 0xFFFFFFFF
堆(Heap)	
未初始化数据段（BSS）	内存低地址 0x00000000
初始化数据段（Data）	
代码段（Code）	

- 假设攻击者能够任意修改或读取内存**数据**，那么想要控制系统的**运行**，一共有哪些攻击方法？
- 系统运行 = 指令 + 数据，一共**只有**以下三种方法：
 - 1) **代码注入攻击**：注入数据，让系统将数据当做指令运行。
 - 2) **代码复用攻击**：控制控制流相关数据，复用系统已有代码（配件），构造配件链进行攻击。
 - 3) **非控制数据攻击**：完全复用系统已有功能和流程，控制关键的控制流无关数据，进行攻击。

- 系统运行 = 指令 + 数据。
- 从动态角度，系统运行过程 = 指令流（控制流） + 数据流。
- 指令流，即**控制流**，是指系统运行过程中指令执行的顺序。
- **控制流相关数据**，是指控制指令执行顺序的相关数据。
 - 跳转指令的目标地址
 - 函数指针，代码指针
 - 函数返回地址

- **控制流劫持攻击**，即需要**改变程序正常执行顺序**的攻击。利用内存漏洞，修改控制流相关的数据，劫持程序控制流，控制系统运行。
- **代码注入攻击**：注入恶意代码，劫持程序控制流，让程序跳转到恶意代码执行，进行攻击。
- **代码复用攻击**：复用系统已有代码（配件），构造配件链。然后，劫持程序控制流，让程序按照配件链的顺序执行，进行攻击。

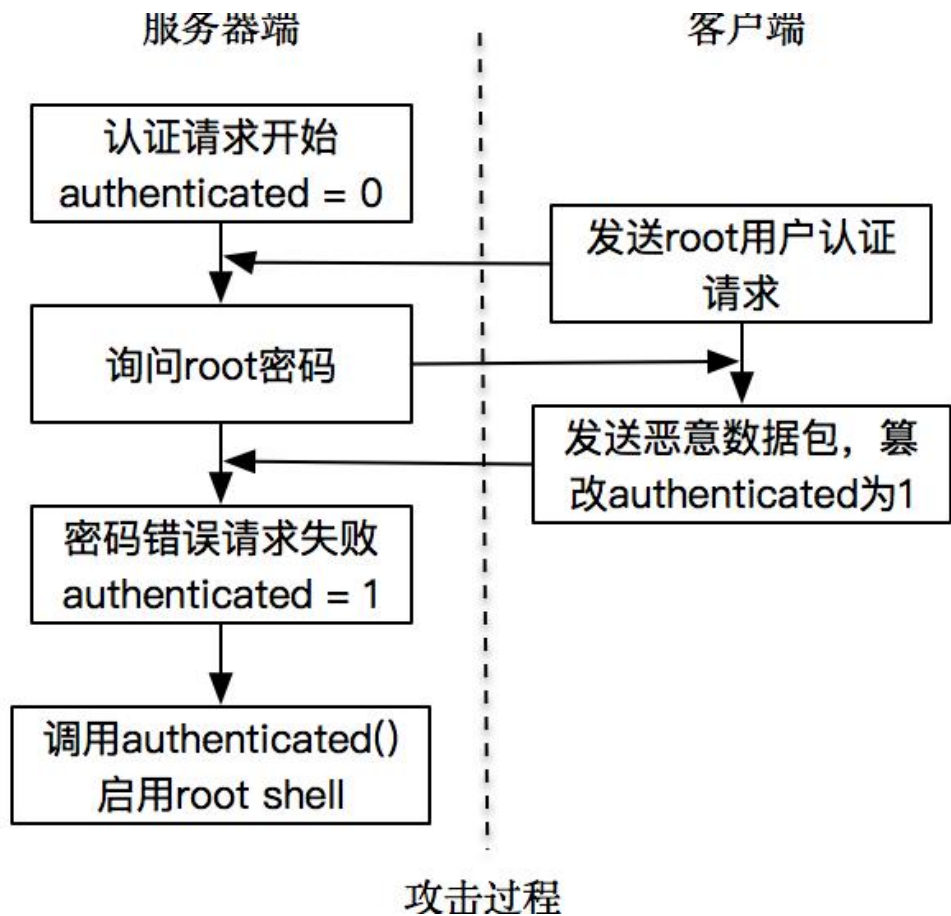
- **非控制数据**，即与控制流无关的数据。
- **非控制数据攻击 (Non-control-data attack)**，不需要劫持控制流，不需要改变程序正常执行顺序，通过修改**控制流无关的关键数据**，实现攻击。

- 控制流无关的**关键**数据，即程序中关键的与安全密切相关的数据，对程序运行过程具有重要的关键性影响。
 - 用户认证数据
 - 系统配置文件数据
 - 用于验证的用户输入的数据
 - 决策数据

攻击样例：利用整型溢出漏洞，篡改系统关键决策数据。


```
void do_authentication(char *user, ...){  
    int authenticated =0;  
    ...  
    while( !authenticated ){  
        type = packet_read(); //漏洞触发点  
        if ( auth_password( user, password ) )  
            authenticated = 1;  
        if ( authenticated ) break;  
    }  
    do_authenticated ( pw );  
}
```

漏洞程序




○防御样例：更改系统关键决策数据的生命周期，缩小攻击面。

```
void do_authentication(char *user, ...){  
    int authenticated =0;  
    ...  
    while( !authenticated ){  
        type = packet_read(); //漏洞触发点  
        if ( auth_password( user, password ) )  
            authenticated = 1;  
        if ( authenticated ) break;  
    }  
    do_authenticated ( pw );  
}
```



漏洞程序

```
void do_authentication(char *user, ...){  
    ...  
    while( !authenticated ){  
        type = packet_read();  
        int authenticated =0;  
        if ( auth_password( user, password ) )  
            authenticated = 1;  
        if ( authenticated ) break;  
    }  
    do_authenticated ( pw );  
}
```



非漏洞程序

- 非控制数据攻击的一个问题：
 - 如果只能控制一些关键数据，是否能够实现任意的恶意操作？
 - 非控制数据攻击是否具有通用性，是否具有学术研究的潜力和价值
- 图灵完备性是评价攻击方法的一个重要指标
 - 图灵完备性说明了该攻击方法的潜力是很大的
 - 在近期工作中，**DOP攻击**证明了非控制数据攻击的图灵完备性，标志着非控制数据攻击正式成为一种新的攻击方法。

○控制流劫持攻击 VS 非控制数据攻击

○控制流劫持攻击（代码注入攻击和代码复用攻击）

- 修改控制流相关数据，如函数指针、跳转指令目标、函数返回地址等

- 改变程序正常控制流

○非控制数据攻击

- 修改关键的和安全相关的和控制流无关的数据

- 不改变程序正常控制流

- 需要攻击者熟知程序语义，利用难度更高

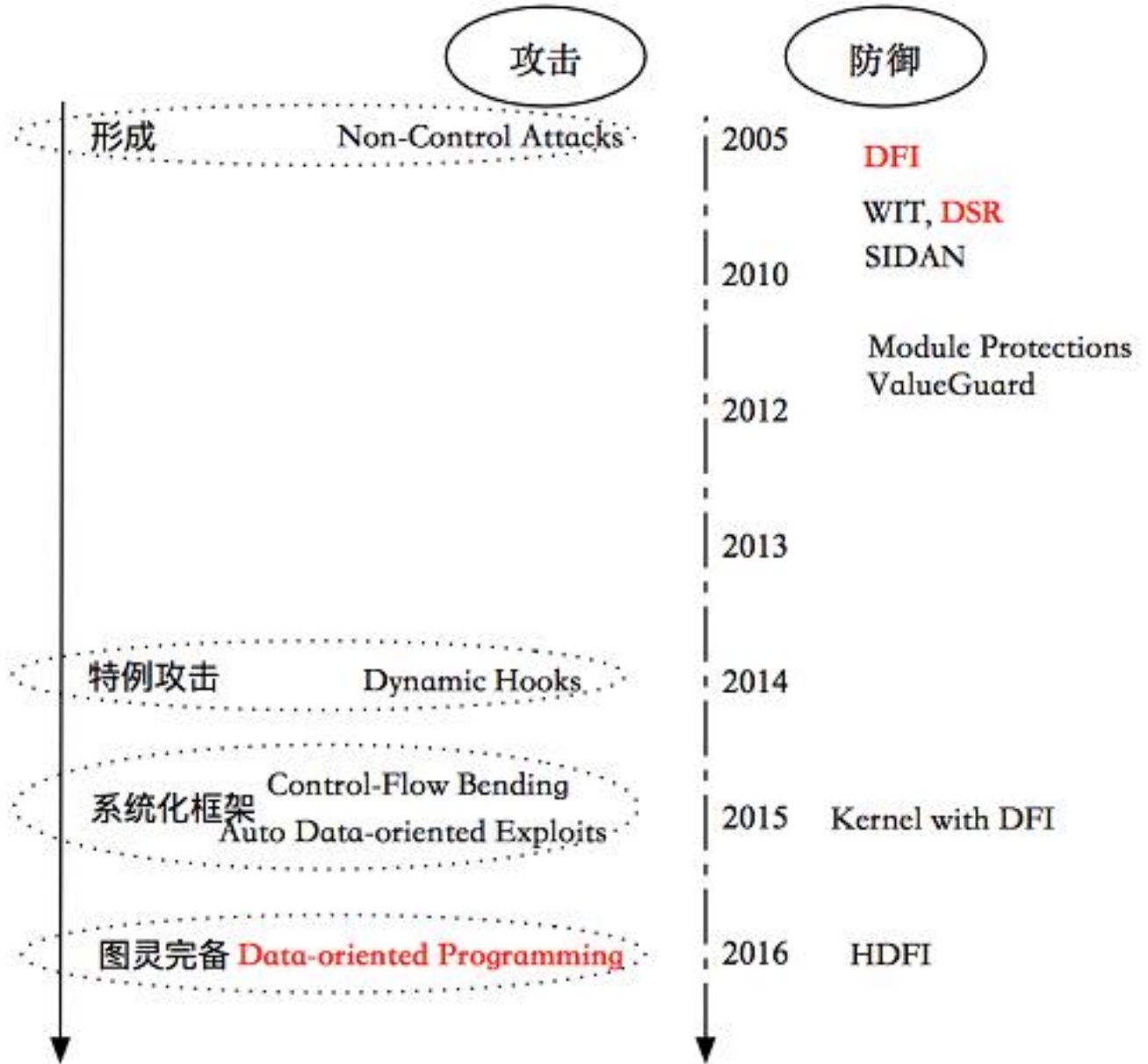
- 非控制数据攻击的**优点**:
 - 完全复用程序正常执行过程，隐蔽性强，难以被发现
 - 能够绕过目前大多数常见的防御机制，如CFI和ASLR等
 - 被证明是图灵完备的攻击
- 非控制数据攻击的**缺点**:
 - 限制很多，利用难度很大

非控制数据攻击的发展历史

- 非控制数据攻击是一种比较新的攻击方法，对其的研究才刚刚开始。
- 最近几年，非控制数据攻击逐渐得到重视。
 - 2005年正式提出了非控制数据攻击的思想。
 - 针对细粒度CFI的缺陷，于2015年提出了CFB，完全符合控制流图CFG，能够绕过最理想情况下的细粒度CFI。
 - 借鉴了ROP的思想，于2016年提出了DOP，是图灵完备的非控制数据攻击方法。

非控制数据攻击及防御发展脉络

非控制数据攻击及防御技术的发展脉络



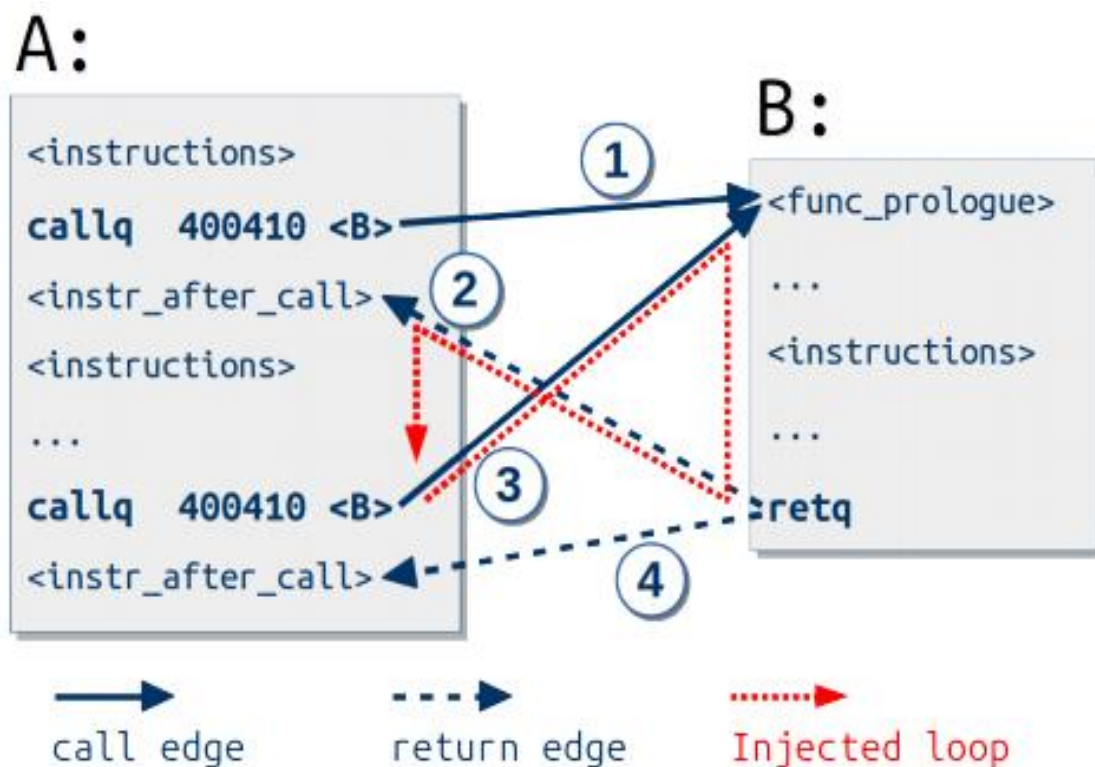
内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

- 细粒度CFI的基本思想是：静态分析所有间接跳转指令的合法跳转目标，获得控制流图CFG，然后要求程序运行时所有间接跳转必须符合CFG。
- 假设能够分析得到**完美的CFG**，能够准确的识别所有间接跳转指令的合法跳转地址。
 - 这在实际系统中是不可能的，只是一种假设。
- 显然，对于**理想的细粒度CFI**，攻击者无法改变程序正常执行顺序，理论上能够**防御所有控制流劫持攻击**。
 - 但是，理想的细粒度CFI是不存在的
 - 完美的CFG是不可能分析得到的

细粒度CFI的缺陷

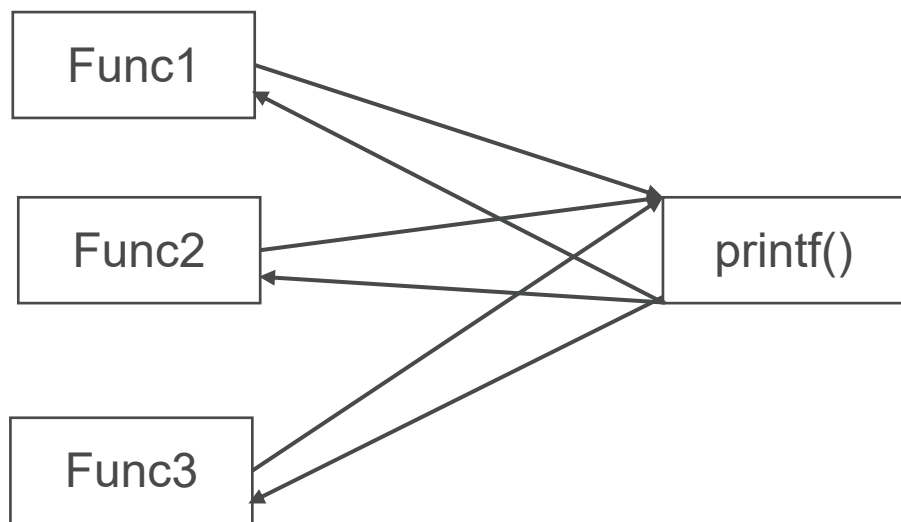
- 细粒度CFI的一个重大缺陷：根据静态分析得到的CFG，无法对程序上下文语义的合法性进行判断。
 - 就算是理想的CFG，也存在精确性不够的问题



- 根据细粒度CFI的缺陷，研究者于2015年提出了一种攻击方法CFB，能够破解理想情况下的细粒度CFI。
- CFB**（Control-Flow Bending，控制流弯曲），USENIX Security 2015。
- 主要思想：
 - 由于细粒度CFI对上下文语义不做限制，所以CFB在不违反CFG的情况下实现攻击。

- 在CFB中，对“non-control-data attack”的定义和本课程有些不同。
- CFB的定义：
 - 如果一个攻击的执行顺序**符合**正常程序的静态**CFG**，则该攻击就是一个non-control-data attack。
- 本课程的定义：
 - 如果一个攻击**没有修改控制流相关数据**，则该攻击就是一个non-control-data attack。
 - 显然，本课程的定义更为严格。
 - 在CFB论文中，作者认为CFB属于非控制数据攻击。但是从本课程的更加严格的定义来看，CFB并不属于严格的非控制数据攻击。

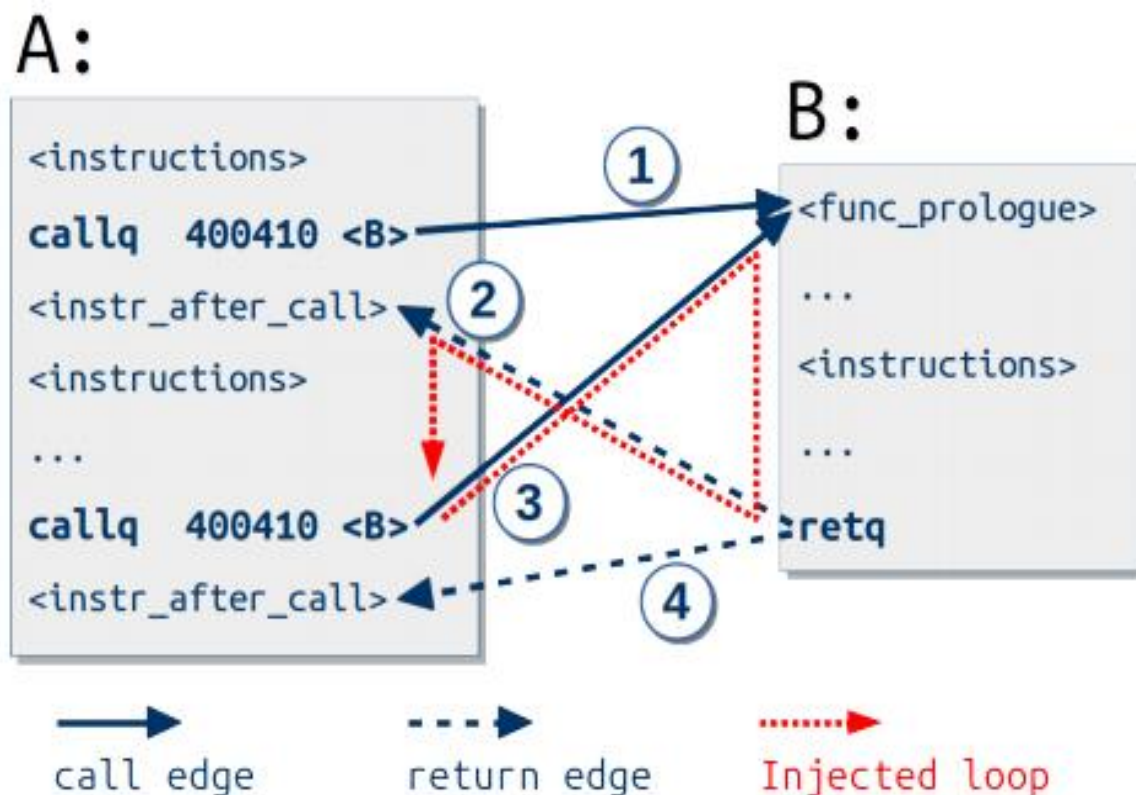
- 对于某些**通用的库函数**，例如printf()，一个程序会在不同的位置多次调用这些库函数。
- 细粒度CFI只以静态分析得到的CFG为判断标准，所以无法对**通用库函数的返回地址**进行精确的检查。
- 因此，在不违反CFG的情况下，攻击者可以通过控制通用库函数的返回地址，实现修改控制流的目的。



○如果只考虑静态分析的CFG，路径1,2,3,4都是合法的。攻击者可以颠倒这些路径的顺序，实现攻击。

○正常路径：1-2-3-4

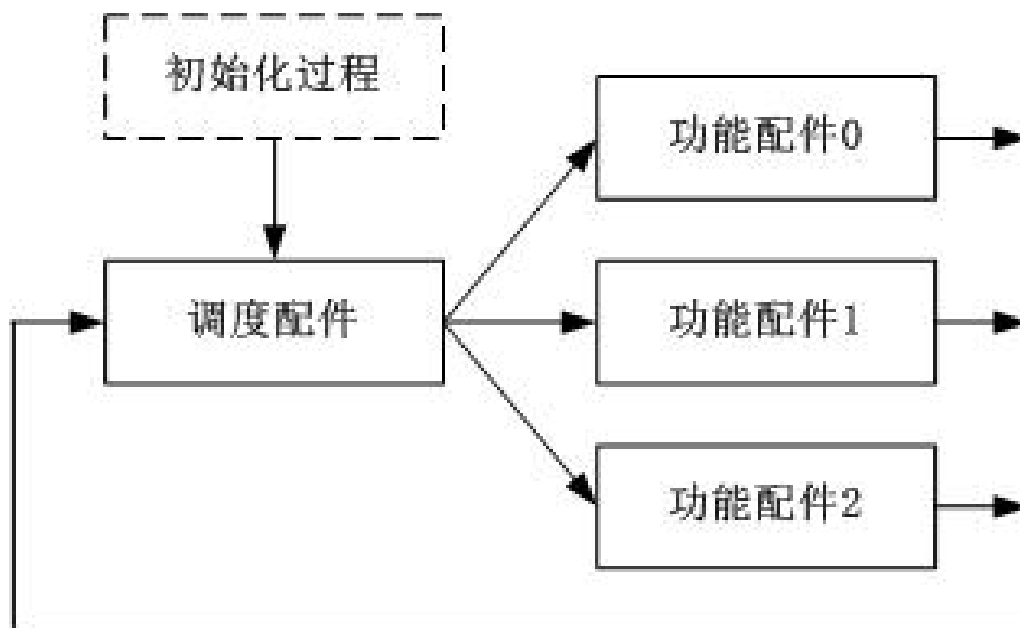
○攻击路径：3-2-3-2



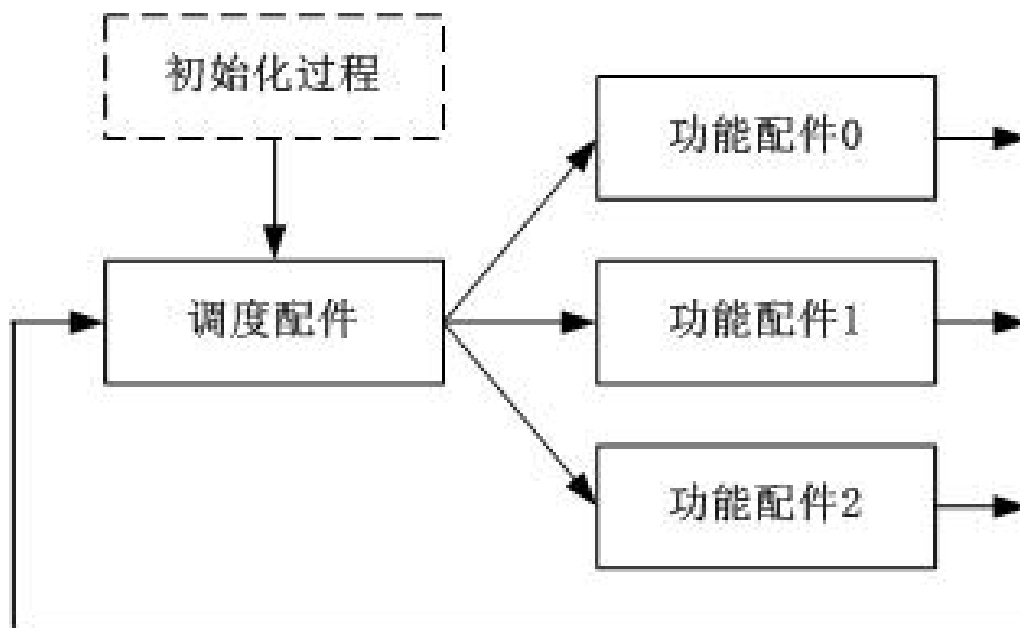
- CFB攻击过程和JOP攻击非常类似，也需要两类配件：
 - **调度配件(dispatcher gadget)**
 - 充当程序EIP的作用，实现控制流的转移。
 - 负责组织功能配件的执行。
 - **功能配件(functional gadget)**
 - 完成某种特定功能的函数配件。

CFB攻击的过程

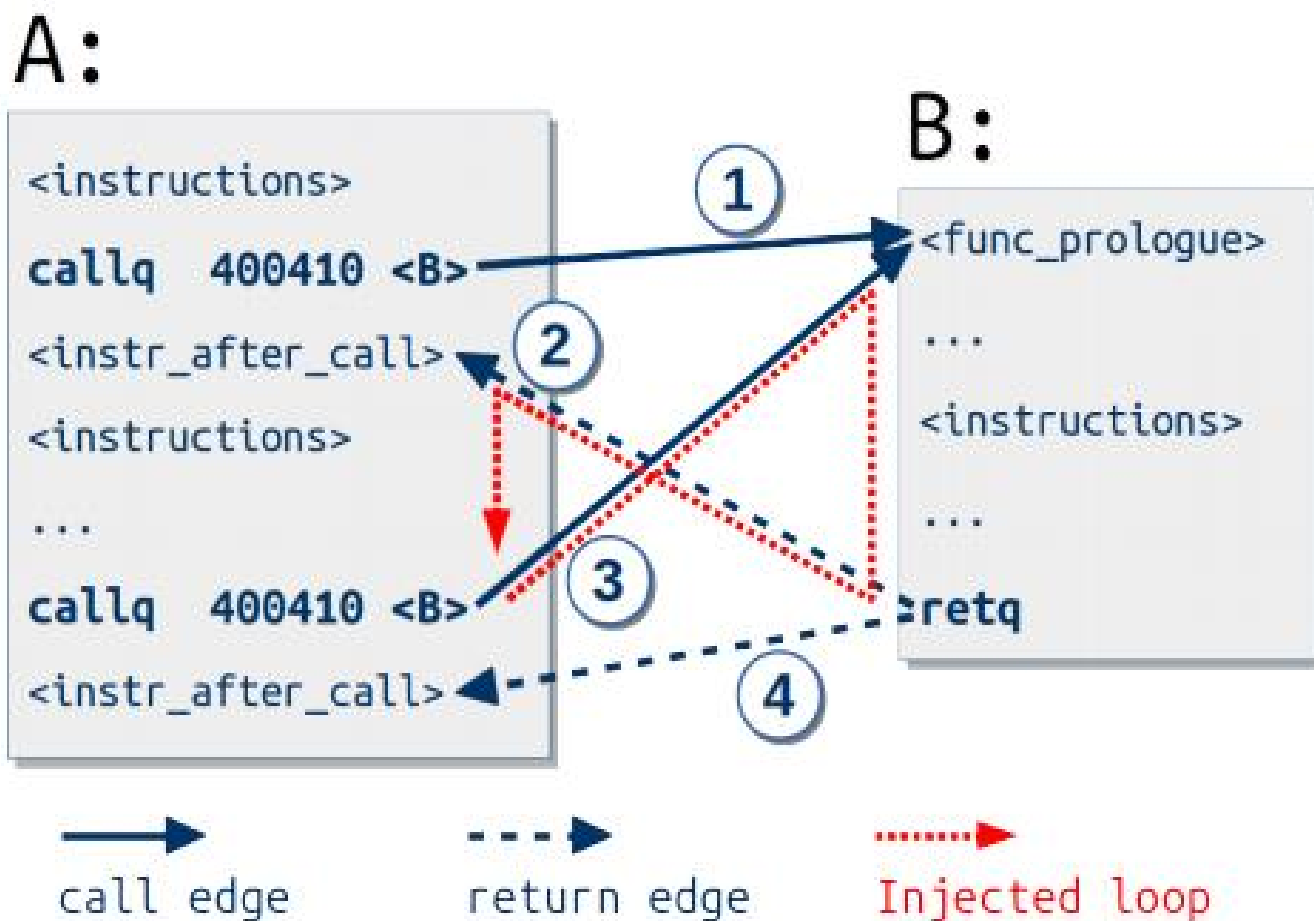
- 初始化CFB攻击，注入攻击数据。
- 控制调度配件的指针，跳转到功能配件。
- 功能配件执行完成，返回到调度配件。
- 调度配件继续，跳转到下一个功能配件。
- 一直循环，完成CFB攻击。



- CFB的调度配件的需求：
 - 需要有一个**循环**。利用调度函数在CFG中创建循环，给予调度函数不同的参数达到控制调度函数返回的效果。
 - 需要有一个指向不同功能配件的**可配置的指针**



- **循环**：一个库函数被多个不同位置调用。攻击者可以在这几个调用点中选择返回目标，形成循环。



- **可配置的指针**：一个能够修改内存地址的库函数。例如，`memcpy()`，`fputs()`，`printf(%n)`。
 - 攻击者利用内存漏洞修改这些库函数的参数，让这些库函数**修改自己的返回地址**。
 - 如果将库函数的返回地址修改为调用该库函数的下一条指令地址，库函数的返回仍然符合CFG的规定，是合法的跳转。

- 功能配件就是实现各种运算操作的配件。
 - CFB的论文给出了一种利用printf函数构造功能配件的方法。
 - CFB功能配件的构造，不局限于printf()，也使用fputs()等其他库函数。
- 具体方法就是，利用printf()函数中的格式化字符串漏洞，实现**基本的逻辑运算**，然后利用这些**逻辑运算组合**就能实现任意的计算操作。

○利用printf实现各种运算操作

○或门：如果两个输入有一个非零，则输出就是非零。

```
void or(int* in1, int* in2, int* out) {  
    printf("%s%s%n", in1, in2, out);  
    printf("%s%n", out, out);  
}
```

○非门：加255相当于减1，只考虑8bit

```
void not(int* in, int* out) {  
    printf("%*d%s%n", 255, in, out);  
    printf("%s%n", out, out);  
}
```


○利用printf实现各种运算操作

○只用一个printf函数实现非门:

```
char* pad = memalign(257, 256);
memset(pad, 1, 256);
pad[256] = 0;
void single_not(int* in, int* out) {
    printf("%*d%s%n%hhn%s%s%n", 255, in, out,
           addr_of_argument, pad, out, out);
}
```

○用于测试一个字节是否等于一个特定的数值

```
void test(int in, int const, int* out) {
    printf("%*d%*d%n", in, 0, 256-const, 0, out);
    printf("%s%n", out, out);
    printf("%*d%s%n", 255, out, out);
    printf("%s%n", out, out);
}
```

- 图灵完备的攻击就是指该攻击能够实现任意的运算操作。
 - 构造图灵完备的攻击是学术研究的一个重要指标
- 构造图灵完备的CFB攻击：
 - 利用printf()构造基本的逻辑运算（与或非）。
 - 然后利用调度配件不断循环调用printf(), 利用基本逻辑运算组合成为不同的计算，最终实现图灵完备的攻击。
 - 与或非是电路的基本单元，计算机系统的所有操作在理论上最终都能转换为与或非的组合

○优点:

- 能够绕过理想情况下的细粒度CFI
- 是图灵完备的攻击
- 在学术上论证了CFI方法存在的理论缺陷

○缺点:

- 只能使用几个常用的库函数
- 虽然能够实现图灵完备攻击，但是需要构造非常复杂的配件链，实现难度很大，复杂性很高，实用性不高

- CFB攻击利用了静态CFG缺少上下文信息的缺陷，尤其是对于常用库函数有多个合法的返回地址。
 - 在实际运行过程中，CFG无法确定哪一个返回地址是合法的。
- 粗粒度CFI中有一个对ret的规则：
 - ret应该跳转到**对应**call指令的下一条指令
- 使用粗粒度CFI的规则，就能够限制CFB利用ret跳转到其他调用位置。

- **影子栈**是实现粗粒度CFI对ret返回地址精确规则的具体实现方式。
- 因此，使用影子栈就能防御CFB的攻击。
- 从另一个角度也可以看出，**粗粒度CFI并不完全是细粒度CFI的子集**。两种CFI有着各自不同的特点。

- 研究者发现了细粒度CFI的一个重大缺陷，然后提出了针对性的攻击方法CFB。
- 研究者认为影子栈是细粒度CFI的重要保障，能大幅提高细粒度CFI的防御效果。
- CFB虽然自称是非控制数据攻击，但是CFB仍然需要修改函数返回地址，也能被细粒度CFI+影子栈所防御。因此，CFB并不属于本课程定义的严格的非控制数据攻击。

内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

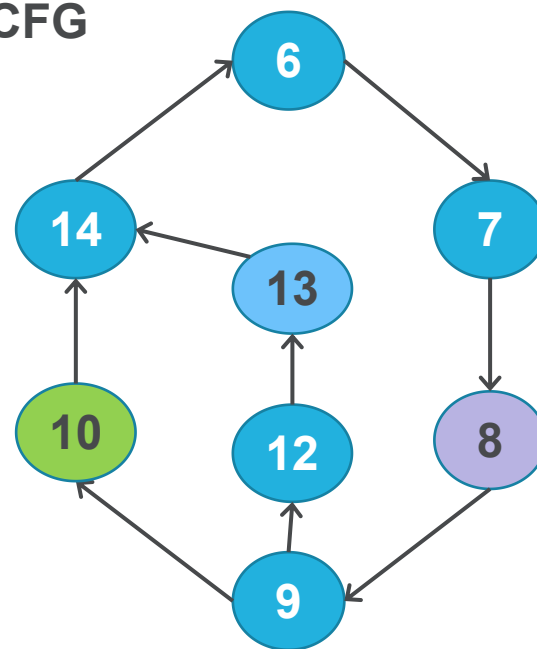
- DOP是非控制数据攻击的**里程碑式**的工作。
- 目前，ASLR和CFI已经提出，并逐渐开始在现实世界中大规模的推广。代码复用攻击逐渐受到了各种防御的限制，实现难度越来越大，复杂性也越来越高。
- 因此，研究者于2016年提出了DOP攻击，能够绕过目前常规的各种防御机制，标志着非控制数据攻击也成为一种主流的攻击方法。

- **DOP** (Data-Oriented Programming, 面向数据的编程方法), 通过内存漏洞修改参与运算的**数据**变量, 在不改变控制流的情况下完成恶意攻击。
- 简单来说, DOP就是标准的非控制数据攻击, 只修改控制流无关的数据, 不修改控制流。

- 一个简单的例子：假设攻击者能够控制cond0-2和para1-5，就可以通过cond0-2选择循环次数和循环内部函数的执行组合，通过para1-5控制函数的参数，实现不同的功能。

```
while(cond0) {  
  if(cond1)  
    func1(para1,para2);  
  else  
    if(cond2)  
      func2(para3,para4);  
    else  
      func3(para5); }
```

CFG



- DOP仍然需要复用系统已有的代码，只是不能打乱重组已有代码的执行顺序。所以，依然将DOP复用的系统已有代码段片段称为**配件**。
 - 由于代码执行顺序不能随意改变，寄存器中存储的数值会经常被其他的无关语句修改。
 - 因此，**DOP将攻击所需的数据保存在内存中**，在每次运算之前才会加载到寄存器中，运算完成后又会马上写回内存，确保数据不被其他无关操作破坏。

- 代码复用攻击的配件：
 - 以**寄存器**为配件之间数据传递的中介
 - 以**间接跳转指令**为配件之间的连接
- DOP的配件：
 - 以**内存**为配件之间数据传递的中介
 - 配件应该包含对内存读写的**访存指令**
 - 配件执行顺序完全按照正常程序执行

○DOP攻击过程和JOP、COOP、FOP类似，也将配件分为两种类型：

- 功能配件**(functional gadget)
 - 完成某种特定功能的配件。
- 调度配件**(dispatcher gadget)
 - 负责组织功能配件的执行。

- 功能配件从内存中读取数据，运算，然后将结果写回内存。

Addition: `srv->total += *size;`

```
1  mov (%esi), %ebx    //load micro-op
2  mov (%edi), %eax    //load micro-op
3  add %ebx, %eax      //addition
4  mov %eax, (%edi)    //store micro-op
```

Load: `*size = *(srv ->cur_max);`

```
1  mov (%esi), %ebx    //load micro-op
2  mov (%edi), %eax    //load micro-op
3  mov 0xb(%ebx), %eax //load
4  mov %eax, (%edx)    //store micro-op
```

内存布局

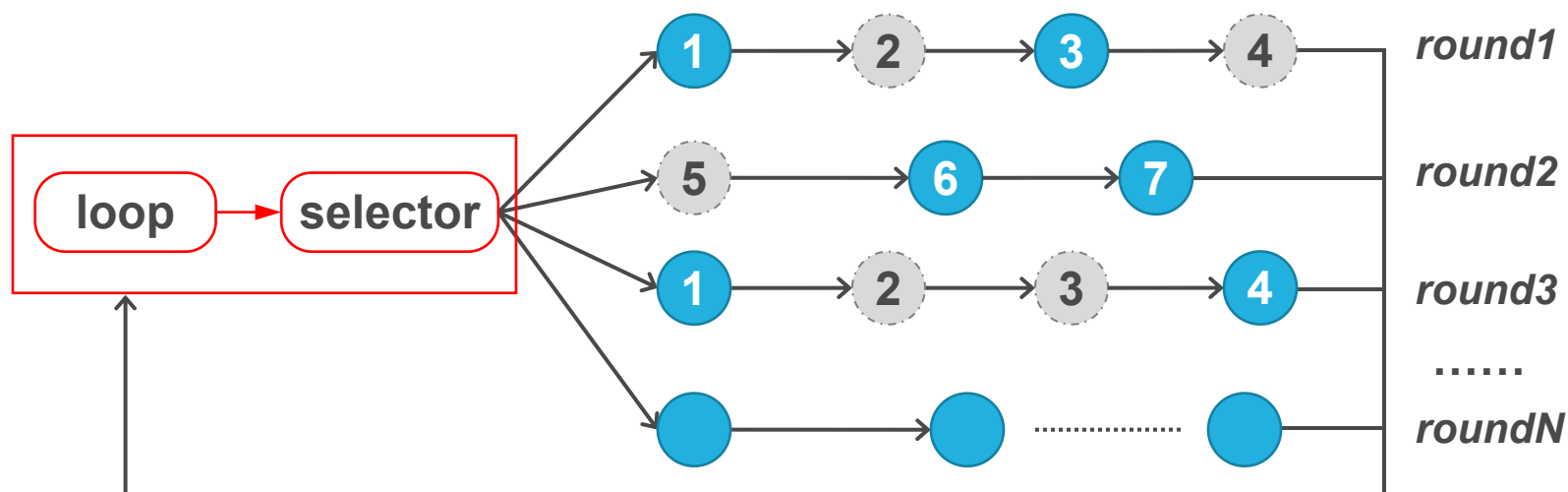


- DOP功能配件的寻找过程：
 - 首先遍历程序中所有的函数，找到store指令，将其视为一个潜在的配件；
 - 然后进一步分析，如果发现在store前面有load操作，则标记将其为可用配件（Data-oriented gadget）。
 - 根据攻击者可控制的变量定义，功能配件有不同的选择优先级：
 - global（全局变量） > function-parameter（函数参数） > local（本地变量）
 - 长度短的 > 长的

○DOP调度配件：

○循环：能够重复的调用配件

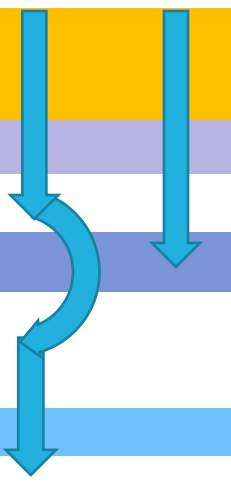
○选择器：选择激活不同的功能配件。选择器可以通过人为交互式输入内存状态来控制功能配件执行。



○DOP调度配件的选择器：

- 根据内存漏洞能够修改不同的分支条件，控制条件跳转的方向，执行处于不同分支的功能配件。

```
6  while (quota-- ) {  
7      readData(sockfd, buf);  
8      if(*type == NONE ) break;  
9      if(*type == STREAM)  
10         *size = *(srv->cur_max);  
11     else {  
12         srv->typ = *type;  
13         srv->total += *size;  
14     } //...(code skipped)...  
15 }
```



- 在程序中找到一个循环，该循环的判断变量可控，并且该循环内部有足够多的可用配件。
- 如果某个函数能够完成特定功能，并且这个函数的参数可以被控制，那么这个函数就是一个潜在的可用功能配件。

Algorithm 2: Gadget dispatcher identification.

Input: G:- the vulnerable program

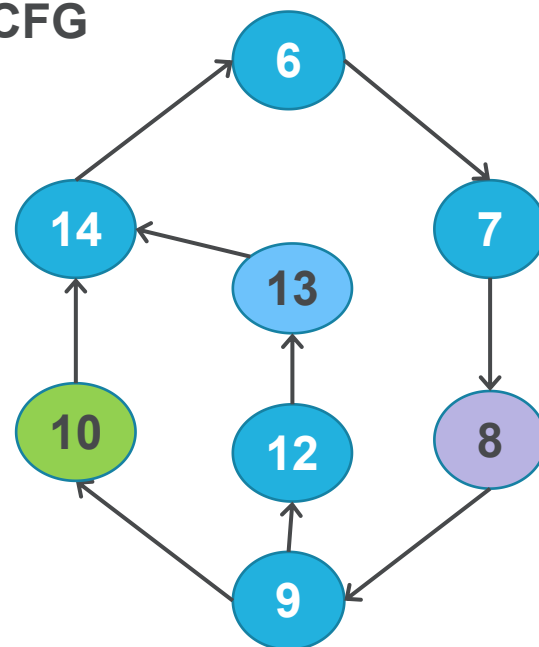
Output: D:- gadget dispatcher set

```
1 D =  $\emptyset$ ;  
2 FuncSet = getFuncSet(G)  
3 foreach  $f \in \text{FuncSet}$  do  
4     foreach  $\text{loop} = \text{getLoop}(f)$  do  
5          $\text{loop.gadgets} = \emptyset$   
6         foreach  $\text{instr} = \text{getNextInstr}(\text{loop})$  do  
7             if  $\text{isMemStore}(\text{instr})$  then  
8                  $\text{loop.gadgets} \cup = \text{getGadget}(\text{instr})$   
9             else if  $\text{isCall}(\text{instr})$  then  
10                  $\text{target} = \text{getTarget}(\text{instr})$   
11                  $\text{loop.gadgets} \cup = \text{getGadget}(\text{target})$   
12             if  $\text{loop.gadgets} \neq \emptyset$  then  
13                  $D = D \cup \{\text{loop}\}$ 
```

- 假设攻击者能够控制所有分支跳转判断参数
- 寻找一个大循环，内部包含很多分支跳转和功能函数
- 利用循环内部分支跳转和功能函数，组合实现恶意攻击

```
while(cond0) {  
  if(cond1)  
    func1(para1,para2);  
  else  
    if(cond2)  
      func2(para3,para4);  
    else  
      func3(para5); }
```

CFG



- 图灵完备的攻击就是指该攻击能够实现任意的运算操作。
- 构造图灵完备的攻击是学术研究的一个重要指标
- 问题：**如何证明一种攻击方法是图灵完备的？

- **问题：**如何证明一种攻击方法是图灵完备的？
- **证明方法：**
- 1) 对计算机系统的操作进行分析，将其分解为最基本的几种操作
 - 例如，与或非，运算+存取+分支跳转
 - 与或非是电路的基本单元，电路的所有操作都可以转化为门电路的组合
 - 指令是软件的基本单元，软件的所有操作都可以转化为指令的组合
 - RISC指令集：所有指令可以分为几种简单的指令，然后由这些简单的指令组合可以完成任意复杂指令的功能

- **问题：**如何证明一种攻击方法是图灵完备的？
- **证明方法：**
- 2) 证明攻击能够实现最基本的几种操作
 - CFB攻击证明能够实现与或非操作
 - DOP攻击证明能够实现最基本的指令功能
- 3) 证明攻击能够实现循环，能够将上述基本操作进行任意的组合，从而证明了攻击的图灵完备性

- 可以用DOP攻击实现六种基本运算操作。
- 使用这六种基本运算操作**组合**就能实现**任意的运算操作**，所以DOP攻击是图灵完备的。

Semantics	Statements In C	Data-Oriented Gadgets in DOP
arithmetic / logical	<code>a op b</code>	<code>*p op *q</code>
assignment	<code>a = b</code>	<code>*p = *q</code>
load	<code>a = *b</code>	<code>*p = **q</code>
store	<code>*a = b</code>	<code>**p = *q</code>
jump	<code>goto L</code>	<code>vpc = &input</code>
conditional jump	<code>if (a) goto L</code>	<code>vpc = &input if *p</code>
p – &a; q – &b; op – 任意的算术/逻辑操作		

○代码复用攻击：

- 以配件为单位复用系统已有代码，**重新组合**，形成配件链，配件链具有新的恶意功能
- 以**寄存器**为配件数据传递中介

○DOP攻击：

- 仍然需要复用系统已有的代码（配件），**不能打乱重组**已有代码的执行顺序
- 以**内存**为配件数据传递中介
- 通过**控制关键数据**，如循环判断、条件跳转判断、函数参数等，控制系统运行，实现攻击

○优点:

- 构建了具有代表性的非控制数据攻击框架
- 隐蔽性强，难以被发现
- 是图灵完备的攻击，不依赖于特定的数据或者函数
- 能够绕过所有针对控制流进行监控的防御机制，包括ASLR和CFI等

○缺点:

- 限制很多，利用难度较高。
- 只能使用程序已有的功能，通过修改参数来控制函数的具体运行。

- DOP是第一个实用的图灵完备的非控制数据攻击，标志着非控制数据攻击成为一种主流的攻击方法。
- 未来，对非控制数据攻击及防御的研究将会成为一个新的热点问题。

内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

- 非控制数据攻击：利用**内存漏洞**，通过**修改控制流无关的关键数据**，实现攻击。
- 对非控制数据攻击的**防御**：
 - 对**内存漏洞的防御（已经介绍过了）**
 - 使用类型安全的高级语言，如Java，Python等。
 - 栈cookie
 - 内存随机化，如ASLR等
 - 防御控制流无关的关键数据被修改
 - DSR
 - DFI

- 对非控制数据攻击的防御思想大多是从对代码复用攻击防御的方法演化而来的。

- DSR:

- 对内存数据空间进行随机化，让攻击者无法修改关键数据。

- 这是从**随机化**方法ASLR发展过来的。

- DFI:

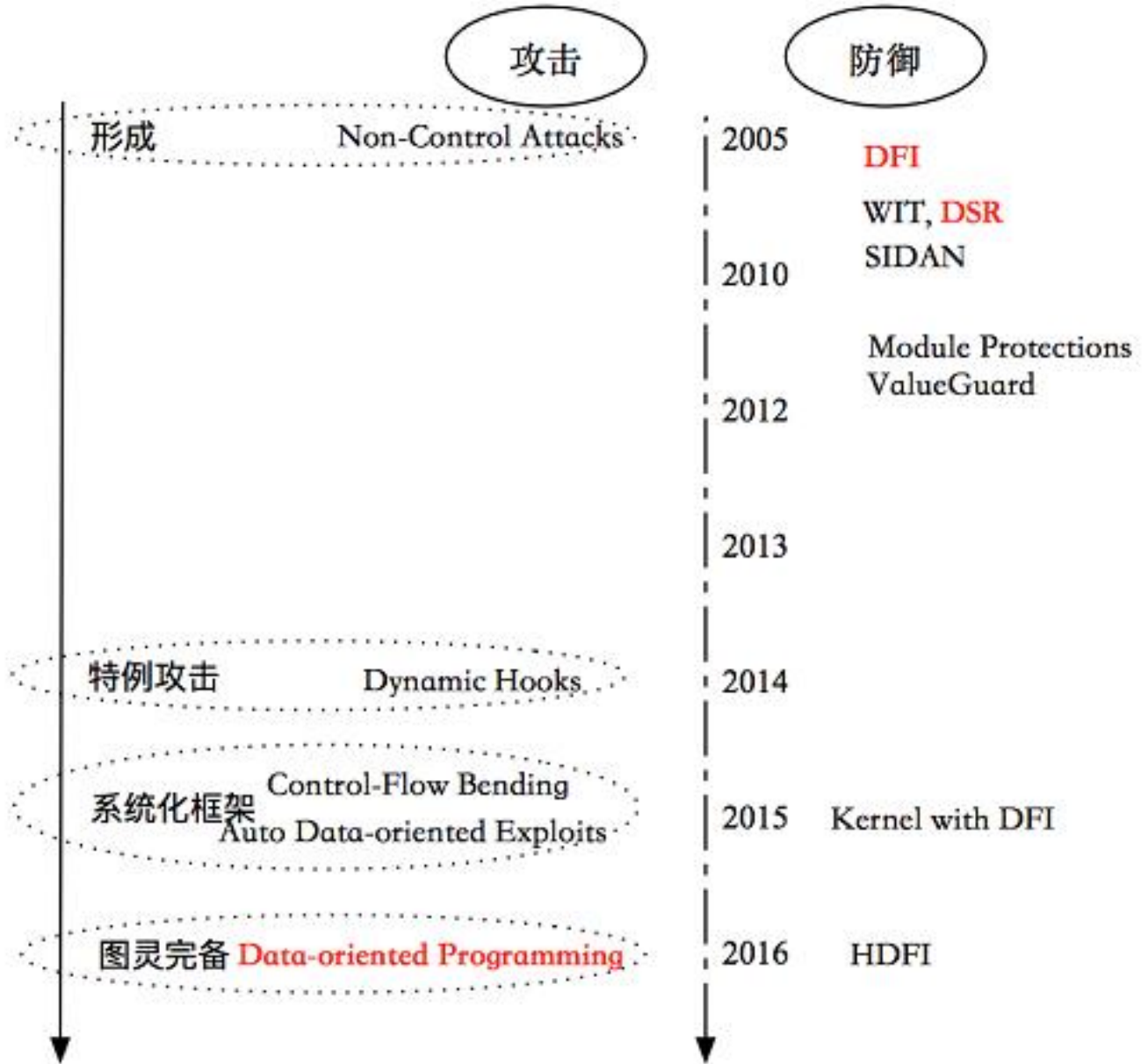
- 静态分析得到数据流图，判断数据读写操作是否符合数据流图。

- 这是从**控制流完整性**CFI发展过来的。

- 非控制数据攻击是一种很新的攻击方法，对其的研究才刚刚开始。
- 同样，对非控制数据攻击的防御的研究也才刚刚开始，相关研究工作目前还很少。
- 到目前为止，还没有针对非控制数据攻击的真正实用有效的防御，还有待未来进一步的研究。

非控制数据攻击及防御发展脉络

非控制数据攻击及防御技术的发展脉络



内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

- **DSR** (Data Space Randomization, 数据空间随机化) , 2008年提出的防御方法。
- 基本原理:
 - DSR把不同的数据对象通过**随机化加密**的方式存储在内存中, 使得数据对象具体的值对攻击者不可知, 从而抵御data-only攻击。
- 实现方法:
 - DSR的随机化加密采用了“**掩码异或**”的思想, 不同的数据对象分配不同且唯一的掩码, 存储前异或加密, 使用前再次异或解密。

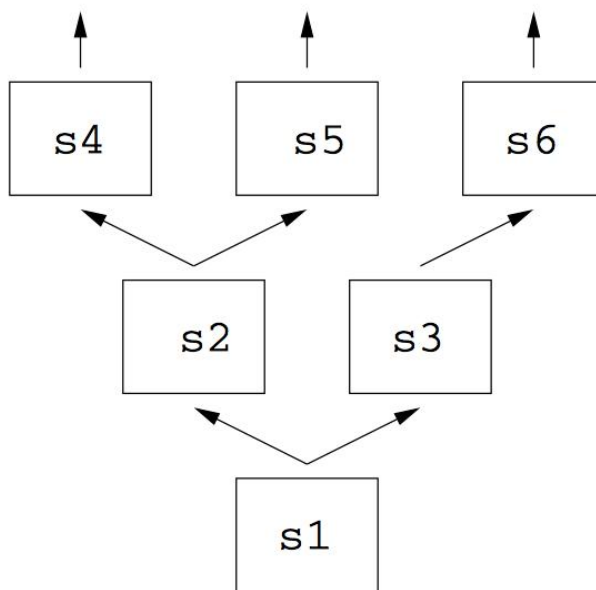
- 掩码异或的加密方式：
 - 对x的正常赋值： $x = v$;
 - 对x的正常使用： $z = x + y$;
 - 对x进行加密： $x = m(x) \oplus v$;
 - 对x的使用： $z = (x \oplus m(x)) + (y \oplus m(y))$;
 - 其中， \oplus 为异或操作， $m(x)$ 为变量x的掩码。
- DSR虽然名字是随机化，但是实际上是通过**加密**的方式对数据进行保护。

○ **源码分析**：通过points-to分析，把变量分成不同的**等价类**。

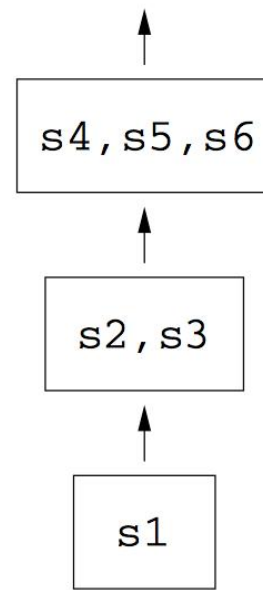
○ **示例(a)**，通过指针s1可以访问对象s2和s3，指针s2可以访问对象s4和s5，指针s3可以访问对象s6，因此可以划分出三个不同等价类，即(c)中所示。

```
s2 = &s4;  
s2 = &s5;  
s3 = &s6;  
  
foo(&s2);  
foo(&s3);  
  
void foo(int **s1)  
{  
    ...  
}
```

(a)



(b)



(c)

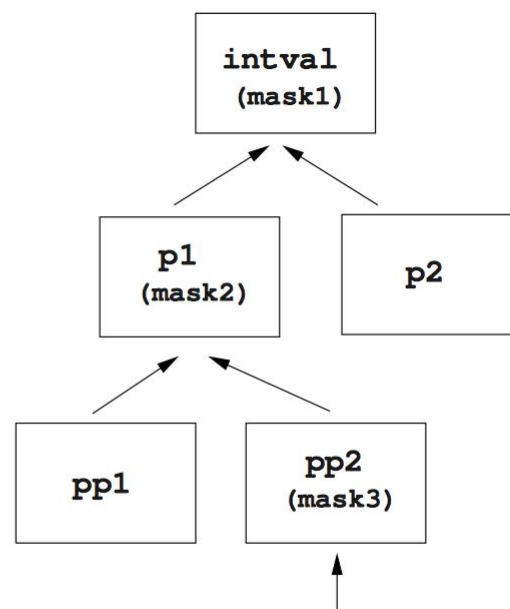
- **掩码分配**：把数据对象划分为不同的等价类。
 - 对于相同等价类的变量，可以分配相同的掩码。
 - 对于不同等价类的变量，使用不同的掩码。
- 例如，对于前面的例子，有三种等价类： $\{s1\}$, $\{s1, s2, s3\}$, $\{s1, s2, s3, s4, s5, s6\}$ ，分别分配不同的掩码。

- **筛选优化**：依据源码分析，把数据对象再细分为溢出候选对象和非候选对象。
 - 对于候选数据对象分配掩码进行加密存储。
 - 对于非候选数据对象按常规方式处理，不进行加密。
- 筛选优化减少了需要加密的数据，能够提高DSR的效率，降低性能损耗。

- 1) 依据图(a)中的源码，通过pointer分析，把不同的数据对象划分为不同层级的**等价类**，如图(b)所示；
- 2) 通过优化策略，**筛选**可能被溢出污染的数据对象候选集，如下图(b)中的pp2, p1, intval；
- 3) 依据(1)和(2)，为数据变量**分配不同的掩码**；

```
int *p1, *p2, **pp1, **pp2, intval;  
...  
int main()  
{  
    ...  
    p1 = &intval;  
    pp1 = &p1;  
  
    pp2 = pp1;  
    p2 = *pp2;  
    ...  
    ... = &pp2;  
    ...  
}
```

(a) A sample C code



(b) Points-to graph for the code

- 4) 使用变量mask1, mask2, and mask3存储(3)中的数据变量的掩码，并通过将掩码初始化为不同的随机数；
- 5) 通过编译器替换需要保护的数据的引用方式。

```
static unsigned int mask1, mask2, mask3;
int **p1_ptr, *p2, **pp1, ***pp2_ptr, *intval_ptr, ...;
int main()
{
    ...
    (*p1_ptr) = intval_ptr;
    (*p1_ptr) = (int *)((unsigned int)(*p1_ptr) ^ mask2);
    pp1 = p1_ptr;
    (*pp2_ptr) = pp1;
    (*pp2_ptr) = (int **)((unsigned int)(*pp2_ptr) ^ mask3);
    p2 = (int *)((unsigned int)(*((int **))
        ((unsigned int)(*pp2_ptr) ^ mask3))) ^ mask2);
    ...
}
static void (__attribute__((__constructor__))) __dsr_init]()
{
    ...
    /* code to allocate memory for intval, p1 and pp2 using their
    pointers intval_ptr, p1_ptr and pp2_ptr respectively. */
    ...
    __dsr_maskassign(mask1); __dsr_maskassign(mask2);
    __dsr_maskassign(mask3);
}
```

○对一些特殊情况的处理：

- 1) **间接引用的数据**：因为静态分析会得到多个可能的值，因此为同一个间接调用的所有可能变量分配同一个掩码；
- 2) **变量同名问题**：同名的不同变量会使用相同的掩码，因此把使用相同掩码的不同变量对象通过页映射机制，分配到相互隔离的内存区域，避免相互干扰。

- 优点:

- 通过加密方式保护关键数据不被篡改

- 缺点:

- 性能损耗大，存在最坏百分之三十的性能损耗

- 实现复杂，实用性不高

- 需要修改程序源码，需要编译器的支持

- DSR是一种以**加密**方式实现**数据随机化**的防御方法，能够防御非控制数据攻击。
 - 通过掩码的形式，对数据进行异或运算，让攻击者无法知道数据的具体值，从而无法读取或修改。
- DSR的实现比较复杂，性能损耗也较高，实用性不高。

内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

- DFI (Data-Flow Integrity, 数据流完整性) , 在2006年被首次提出。
- 首先通过分析正常程序行为, 得到正常程序的数据流图。然后分析当前程序行为, 判断当前程序的数据流是否符合正常。如果不符合, 则认为发生了异常, 需要终止程序运行。
- DFI和CFI的基本思想是一样的。
 - CFI: 以静态分析的控制流图为异常判断标准。
 - DFI: 以静态分析的数据流图为异常判断标准。

- 静态分析
 - 标记所有变量的赋值与使用，生成每个使用的合法赋值集合，构建数据流图 (Data-Flow Graph, DFG)
- 动态监控
 - 对于每次赋值，更新对应标记为赋值标记
 - 对于每次使用，检查是否合法
- 触发警报处理

- 1) 找到程序中所有的写指令 (store, push, call) , 即所有会修改内存数据的指令
- 2) 给每一个写指令都附上一个唯一的标签
 - 标签是程序的行号
- 3) 找到程序中所有的读指令 (load, pop, ret) , 即所有会读取内存数据的指令
- 4) 分析每一个读指令可能读取的内存区域, 将所有会修改该内存区域的写指令找到, 将这些写指令的标签集合赋给该读指令
 - 一个读指令对应一系列写指令的标签集合

○代码

- `A=C` #0
- `B=read_string()` #1
- `If A>10:{` #2
- `...}`
- `Else:{`
- `...`
- `}`

○DFG (Data Flow Graph)

- 根据静态分析，#2的使用只应该来源于#0。
- 如果#1发生缓冲区溢出覆盖了A，那么A就被标记为#1。
- 一旦当A被使用时，发现标记是#1，触发警报。

- 通过**插桩**的方法实现对每条内存读写指令的监控。
 - RDT(runtime definitions table): 记录每一个内存位置最后一次写指令的标签。
 - 对每一条写指令插桩, 动态更新RDT。
 - 执行写指令时, 将写指令对应的标签写入RDT
 - 读指令从RDT取回标签, 检测标签是否符合DFG。
 - 执行读指令时, 从RDT取回对应内存位置的标签, 将该标签和读指令对应的标签集合对比, 判断标签是否在集合内部。

- DFI是目前防御非控制数据攻击的最主要方法之一。
- 但是，DFI仅仅是一个学术上的研究，还没有在实际系统中被应用。
 - DFI实现过于复杂，实用性不高。
 - 需要静态分析，获得控制流图DFG。
 - 需要二进制插桩，修改二进制文件。
 - DFI性能损耗极大，需要实时监控和分析每一个读写操作是否合法。

- 优点:

- 抓住了非控制数据攻击的本质特征，防御效果很好。

- 缺点:

- 实现复杂，性能损耗过高，因此实用性不高，没有被真实系统采用（实用性问题）

- 难以生成一个精确的DFG（精确性问题）

- 完整性保护的一大前提是：先**静态分析**获得合法的路径，然后以此为依据判断程序行为是否合法，是否异常
 - CFI控制流完整性，需要分析得到CFG控制流图
 - DFI数据流完整性，需要分析得到DFG数据流图
- 然而，静态分析存在**精确性问题**
 - 绝对精确的CFG和DFG是不可能得到的
 - 就算是相对精确的CFG和DFG也很难得到
 - 当前现实中，分析得到CFG和DFG是非常不精确的，存在很大的误报和漏报
- 当前一种常用的CFG分析方法：参数匹配
 - 当调用者和被调用者的**参数数量及格式完全匹配**，就认为这个调用是合法的
 - 显然，该方法是非常不精确的

○静态分析精确性不足带来的问题

- 由于CFG和DFG的精确性不足，因此存在合法的路径被当做非法的路径，或者非法的路径被当做合法的路径
- 通常，为了让程序正常执行，不让程序随意崩溃，当前CFI和DFI会尽量避免错误的判断（**将合法行为当做非法行为**），因此**对合法行为的判定标准非常宽松**，所以会存在非常多的漏报（**将非法行为当做合法行为**）

○于是，攻击者可以利用以上问题，采取针对性的攻击

- CFB（Control-Flow Bending，控制流弯曲攻击）：利用CFG的不精确性，可以在不违反CFG的基础上实施攻击。
- 同样，DFB（Data-Flow Bending，数据流弯曲攻击，2019年提出）：利用DFG的不精确性，可以在不违反DFG的基础上实施攻击。

- 采用更先进的静态分析方法，可以提升静态分析的精确性，提升CFG和DFG的精确性
 - 域敏感的静态分析
 - 流敏感的静态分析
 - 路径敏感的静态分析
 - 上下文敏感的静态分析
- 但是，**绝对精确是永远达不到的**，因此，CFI和DFI始终存在被攻破的可能性
 - 然而，随着精度的提升，**攻击的难度也会越来越高**，攻击的成本也越来越高，从而可以在一定程度上缓解攻击的威胁

○DFI性能损耗高的原因：

○需要对每一个读写操作进行额外的操作，性能损耗很高

- 当执行内存写指令时，需要将此指令的标签写入指令所访问数据对应的RDT表中，因此每条内存写指令都会产生一次额外的内存访问。
- 当执行内存读指令时，需要从内存中读取指令所访问数据对应的标签，并与数据对应的合法标签进行比较，因此每条内存读指令也会产生一次额外的内存访问和一次额外的比较操作。

○需要为每一个字分配标签，存储占用很高

○最核心的原因：**每一个读写操作都会带来额外的操作**

○DFI性能优化思路：减少读写操作带来的额外操作

○1) 对额外操作进行优化，提升执行的效率

○TMDFI：采用**硬件支持**来加速对读写操作的检查

○2) **大部分**读写操作不需要进行额外的操作，只需要对**部分**读写操作进行检查

○WIT：只对**写操作**进行检查

○KDFI：只对**关键数据**的读写操作进行检查

○3) **不对读写操作进行检查，而采取其他防御方案**

○HDFI：采用**隔离**的思想，将内存**分区**，不同区域之间不能直接进行数据交互

- TMDFI (Tagged Memory Supported Data-Flow Integrity)
- 原理：
 - 利用专门的硬件功能，支持DFI对内存读写操作的检查
- 优点：
 - 采用硬件支持加速，DFI的效率变得更高，性能损耗降低
- 缺点：
 - 仅仅是硬件加速，在方法上没有创新和改进

○WIT (Write Integrity Testing, 写完整性检查)

○原理:

- 对同一对象的写指令分配相同的标签, 在内存写指令之前进行检查, 判断针对数据的内存写是否合法。

○优点:

- 一个基本前提: **内存写操作的数量要远小于内存读操作**
- 所以, WIT只对内存写操作进行检查, 而不检查内存读操作
- 相对于DFI, **WIT需要检查的指令数量更少**, 所以效率更高
- 此外, 确保内存写操作的安全性, 能避免攻击者篡改数据, 从而**能够防御控制流劫持攻击和非控制数据攻击**

○缺点:

- 无法阻止非安全的内存读, 可能会产生**内存信息泄露攻击等**

○KDFI (Key Data Flow Integrity, 关键数据流完整性)

○原理:

- 攻击者往往只会攻击少量关键的数据, 因此**只需要保护少量关键数据**即可, 其他非关键数据没必要保护
- 所以, 对数据进行划分, 分为关键数据和非关键数据

○优点:

- 关键数据占整个数据的**比例很小**, 只对关键数据进行保护, 产生的性能损耗较低
- 关键数据是**攻击必要的数据**, 如果不能篡改关键数据, 则攻击无法成功, 因此能够防御各种可能的攻击, 安全性和DFI相当

○缺点:

- 关键数据的定义和划分不明确

- HDFI (Hardware-assisted data-flow isolation, 硬件辅助的数据流隔离)
- 原理：
 - 属于安全隔离技术的一种，注意和完整性保护方法的区别
 - 将内存划为敏感区和非敏感区，非敏感区的读写操作不能直接访问敏感区的数据
- 优点：
 - 实现较为简单，性能损耗较小
- 缺点：
 - 划分的粒度过粗
 - 敏感区内部的攻击无法防御，安全性不足

- 本节介绍的DFI是防御非控制数据攻击的主要方法之一。
- 但是，DFI实现复杂，性能损耗极大，实用性不高。
- 和CFI一样，原始的DFI只是提出了数据流保护的思想，实用性不高。
- 因此，研究者进一步对DFI进行优化，提高效率，降低复杂度和性能损耗，使优化后的DFI成为一种可实用的防御技术。

内容概要

- 非控制数据攻击
 - CFB
 - DOP
- 对非控制数据攻击的防御
 - DSR
 - DFI
- 总结

- 主要介绍了**内存漏洞和运行时安全**。
 - 内存漏洞详解（漏洞）**
 - 栈漏洞
 - 堆漏洞
 - 内存信息泄露等其他类型漏洞
 - 运行时安全（攻击和防御）**
 - 代码注入攻击及防御
 - 代码复用攻击及防御
 - 非控制数据攻击及防御

- 系统运行 = 指令 + 数据
- 软件安全：
 - 直接控制指令：
 - 恶意程序，如病毒、木马等。
 - 注入可执行的代码和命令，如恶意脚本、SQL注入等。
 - 只可以修改数据：
 - 代码注入攻击：将数据当做指令运行。
 - 代码复用攻击：复用系统已有代码，改变控制流。
 - 非控制数据攻击：复用系统已有代码，不改变控制流。

- 基本的假设：攻击者总是可以控制数据。
 - 系统必然需要输入数据。
 - 软件漏洞是不可避免的，而内存漏洞属于软件漏洞的一种。
 - 攻击者能够利用内存漏洞，修改或读取内存中的数据。
- 以上假设是符合实际情况的。

- 在内存漏洞存在的前提下，研究如何防御内存漏洞引发的各种攻击。
 - 隔离，随机化，加密：通过种种方法，让攻击者无法获取或修改需要的数据。
 - 异常行为检测，完整性分析：根据攻击行为和正常行为的区别，判断是否发生攻击。
- 安全研究的目标：增加攻击者发现漏洞和利用漏洞进行攻击的难度，让实施攻击的**成本**大于攻击得到的利益。

- 大部分安全问题都可以由软件自行解决。
- 还有一些安全问题主要由硬件来解决：
 - 硬件漏洞
 - 系统启动安全
 - 软件实现性能损耗太高，需要底层硬件支持
- 本课程想要进行的研究：属于体系结构安全研究领域，需要用**软硬件结合方式解决的常见的**安全问题：
 - 由内存漏洞引发的运行时安全

- 软硬件结合的安全方案：需要**硬件、操作系统、编译器**等**计算机体系结构**各个不同层次的支持
- 以不可执行位保护为例
 - 操作系统：需要在页表中增加不可执行位，需要在程序运行时管理每个内存页面的不可执行位。
 - 硬件：需要在处理器中增加对不可执行位的判断逻辑。
 - 可执行文件：需要在文件的代码区标记可执行，在文件的数据区标记不可执行。
 - 编译器：在编译生成可执行文件时，需要生成不可执行的标识。

○非控制数据攻击

○**DOP**: Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks, S&P 2016

○对非控制数据攻击的防御

○**DFI**: Securing software by enforcing data-flow integrity, OSDI 2006

○课程总结汇报+课堂开卷考试

○课程总结汇报：

○时间：第16周周二下午5-7节（6月11日，13点30分-16点20分）

○汇报要求：制作汇报PPT并上台演讲，每人5分钟

○汇报内容：

○汇报PPT**不要讲**课堂讲授的知识，**也不要讲**自己调研或实验的**常识性**内容

○汇报PPT**应该讲**学习本门课程的个人体会和感悟，调研到的新颖的、有价值、有意思的内容，做实验遇到的问题和解决方法、实验技巧等

○也可以提一些对课程的意见和建议

○总之，**不要讲大家都**知道的东西，**要讲一些大家不知道的、对大家有帮助、或大家感觉到有意思的内容**

○就像是写一篇论文，要有**创新性和价值**，不能是已有的、重复的内容

○课程总结汇报+课堂开卷考试

○课堂开卷考试：

○时间：第16周周日晚上9-11节（6月16日，18点10分-21点）

○地点：教一楼108

○少数几位周二下午有课的同学，在本次课进行汇报

○等所有同学全部汇报结束后，开始考试

○考试内容：

○开卷

○开放性问题

○考试题目主要来自于大家平时的作业和课程讲授内容

Q&A