内存破坏漏洞的防护

- 一直是研究的热点
  - 软件保护
  - 系统防护

- 进20年的研究进展（一点点）
  - 获取权限的方法仍未改变
  - 漏洞利用的技术越来越高级
  - 从栈转移到堆区

控制流劫持的防御

- Bug是发生劫持的根本原因
- 利用分析工具查找软件bug
- 证明/验证软件的正确性

- 攻击缓解技术（对程序透明化的保护技术）
- Canaries
- DEP(数据执行保护)/NX(禁止运行)
- ASLR(地址空间布局随机化)

漏洞利用阻止措施

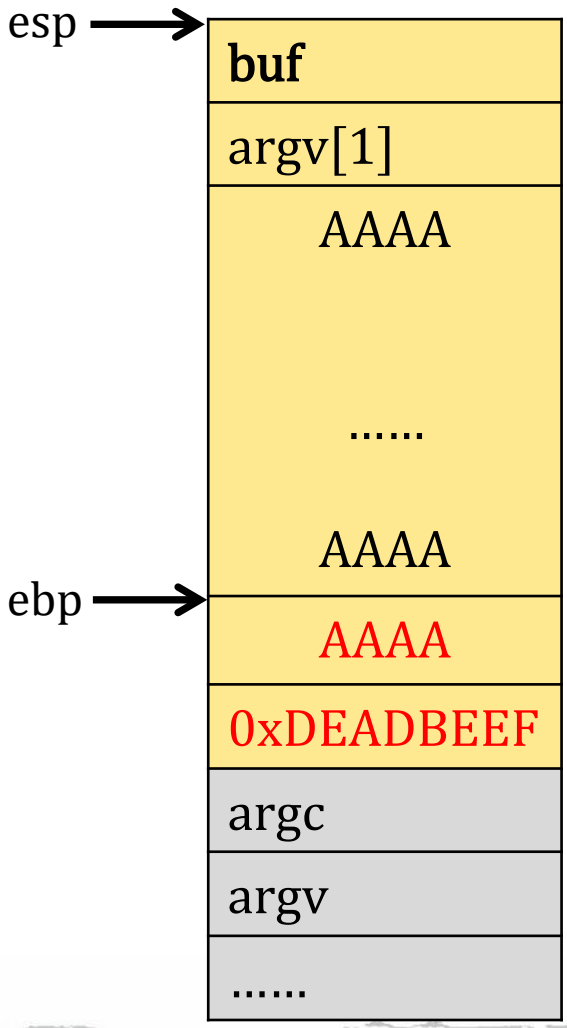
- 漏洞利用条件：
  - 溢出 [返回地址可被修改]
  - 控制权能够顺利转交给修改后的返回地址（控制权获取）
  - 被修改的返回地址有效（指向可执行代码）
  - ShellCode代码可执行（数据）
  - ShellCode代码可以进行恶意操作
    - 下载执行
    - 远程Shell等

Canary / Stack Cookies

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

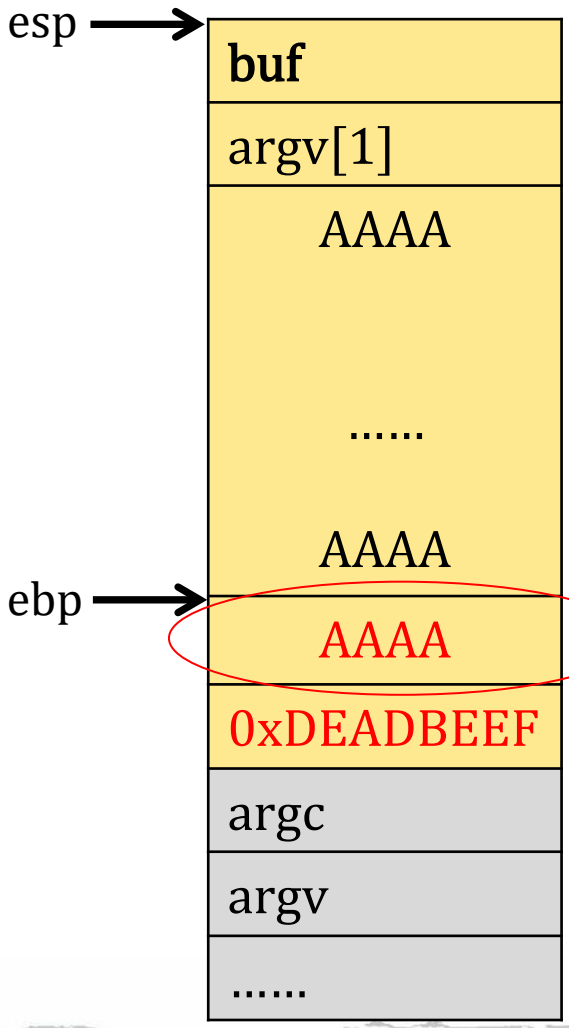
输入“A”x68“\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



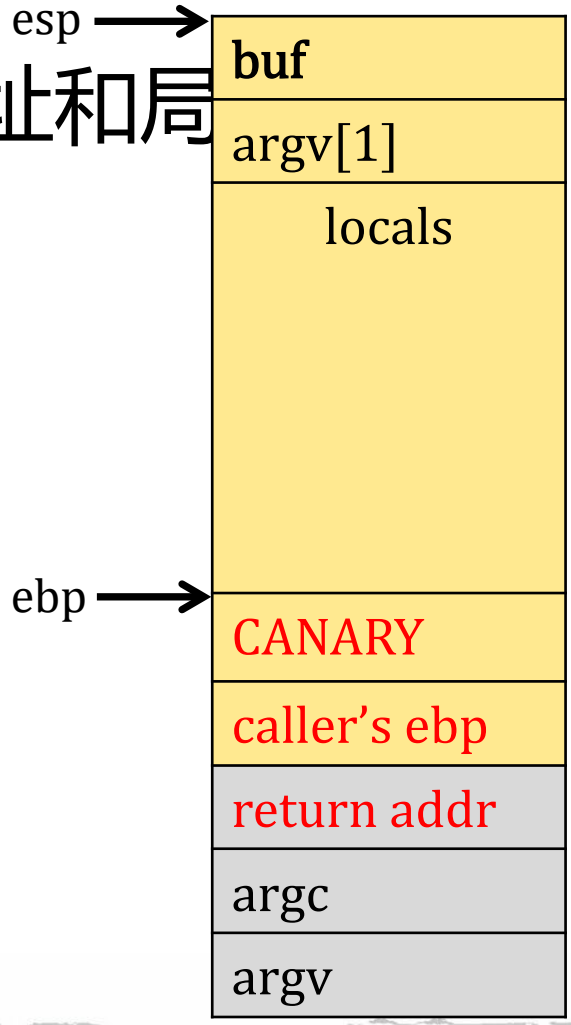
输入“A”x68. “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



StackGuard [cowen etal 1998]

- 核心思想
- 将canary(探测值)插入到返回地址和局间
  - 随机性确保
- 在函数返回之前检查canary
- 错误的canary→溢出



Stack Cookie的实现-GS

- 系统以.data节的第一个DWORD作为Cookie的种子，或称原始Cookie（所有函数的Cookie都用这个DWORD生成）

```
00401003   83EC 20          sub     esp, 20
00401006   A1 00304000      mov     eax, dword ptr [403000]
0040100B   33C5             xor     eax, ebp
0040100D   8945 FC          mov     dword ptr [ebp-4], eax
00401010   CC               int3
00401011   8B45 08          mov     eax, dword ptr [ebp+8]
00401014   8945 EC          mov     dword ptr [ebp-14], eax
00401017   8D4D F0          lea     ecx, dword ptr [ebp-10]
0040101A   894D E8          mov     dword ptr [ebp-18], ecx
0040101D   8B55 E8          mov     edx, dword ptr [ebp-18]
00401020   8955 E4          mov     dword ptr [ebp-1C], edx
00401023   8B45 EC          mov     eax, dword ptr [ebp-14]
00401026   8A08             mov     cl, byte ptr [eax]
00401028   884D E3          mov     byte ptr [ebp-1D], cl
0040102B   8B55 E8          mov     edx, dword ptr [ebp-18]
0040102E   8A45 E3          mov     al, byte ptr [ebp-1D]
00401031   8802             mov     byte ptr [edx], al
00401033   8B4D EC          mov     ecx, dword ptr [ebp-14]
00401036   83C1 01          add     ecx, 1
00401039   894D EC          mov     dword ptr [ebp-14], ecx
0040103C   8B55 E8          mov     edx, dword ptr [ebp-18]
0040103F   83C2 01          add     edx, 1
00401042   8955 E8          mov     dword ptr [ebp-18], edx
00401045   807D E3 00       cmp     byte ptr [ebp-1D], 0
00401049 ^ 75 D8            jnz     short 00401023
0040104B   B8 01000000      mov     eax, 1
00401050   8B4D FC          mov     ecx, dword ptr [ebp-4]
00401053   33CD             xor     ecx, ebp
00401055   E8 23000000      call    0040107D
0040105A   8BE5             mov     esp, ebp
0040105C   5D               pop     ebp
```

Stack Cookie的实现-GS

- 在栈帧初始化以后系统用ESP异或种子，作为当前函数的Cookie，以此作为不同函数之间的区别，并增加Cookie的随机性
- 在函数返回前，用ESP还原出Cookie的种子
- 最后，调用Security_check_cookie 函数进行校验

Stack Cookie的实现-GS

- 强制GS：
  - #param strict_gs_check(on) 选项可以对任意类型函数添加Security Cookie
- VS2005后采用变量重排技术：将字符串变量移动到栈帧的高地址

```
#include"stdafx.h"
#include"string.h"
#pragma strict_gs_check(on) // 为下边的函数强制启用 GS
intvulfuction(char * str)
{
        chararry[4];
        strcpy(arry,str);
        return 1;
}
int_tmain(intargc, _TCHAR* argv[])
{
        char* str="yeah,i have GS protection";
        vulfuction(str);
        return 0;
}
```

Stack Cookie的实现-GS

- 为性能考虑，GS默认不应用于以下情况：
  - 函数不包含缓冲区
  - 函数被定义为具有变量参数列表
  - 函数使用无保护的关键字标记
  - 函数在第一个语句中包含内嵌汇编代码
  - 缓冲区不是8字节类型且大小不大于4个字节

Cookie无法防御的部分

- 未被保护的函数
- 针对基于改写函数指针的攻击，如C＋＋虚函数攻击
- 针对异常处理机制的攻击
- 堆溢出攻击

Canary绕过方法

- 伪随机的猜测
- 通过任意地址写漏洞直接写返回地址，而不是stack smash
- 特殊情况下，可以劫持canary检测失效的处理函数
  - 如stack_check_fail函数

Canary评价

性能	• 每个函数都需要若干指令 • 时间：平均几个百分点
部署	• 重编译即可；无需改变代码本身
兼容性	• 完整兼容，对外透明
安全保障	• 并无安全保障

Data Execution Provention(DEP)/ No eXecute(NX)

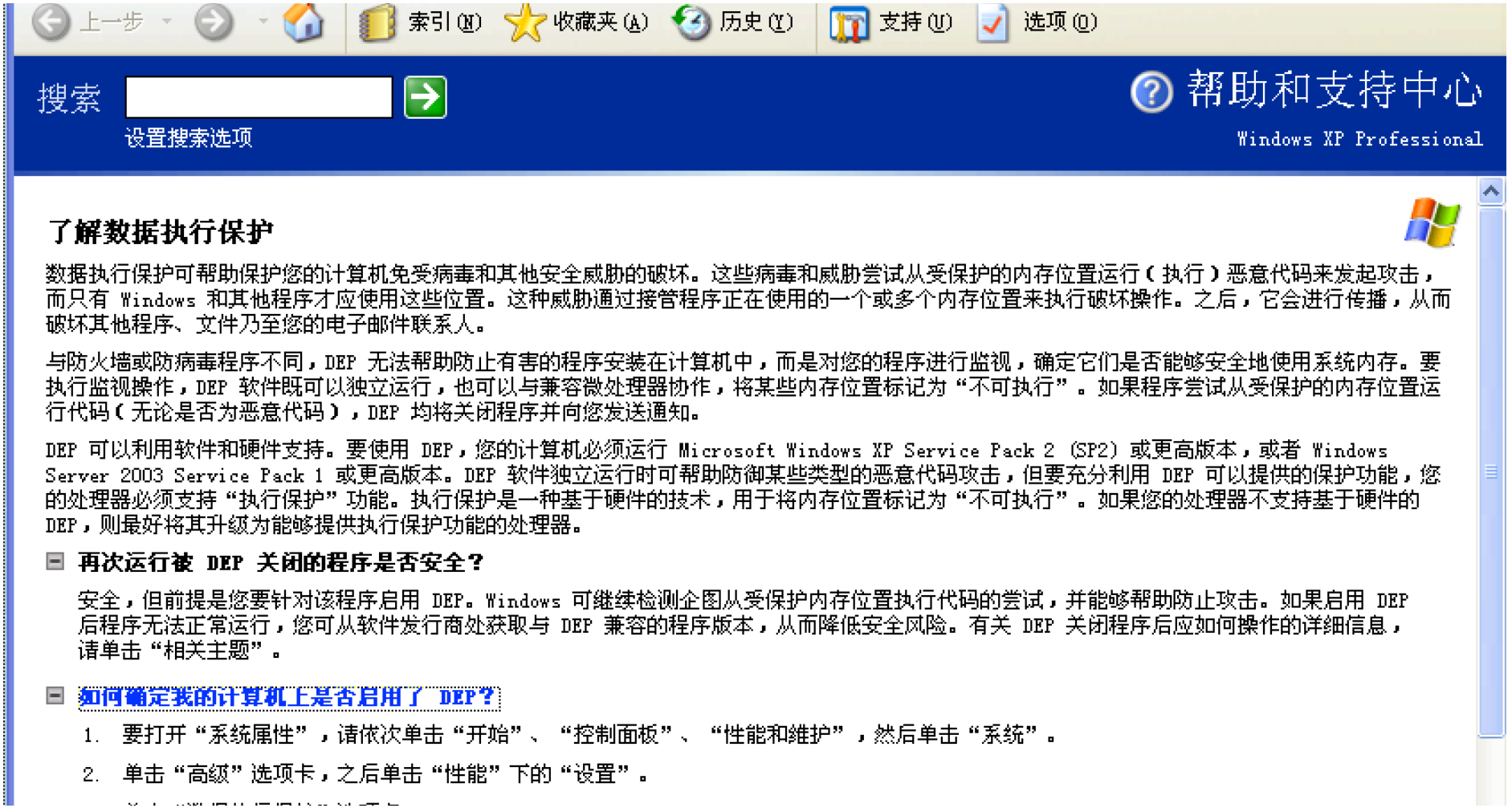
DEP的思想

- 最典型的缓冲区溢出漏洞是把shellcode注入到栈上
  - 栈其实是用来保存函数运行时的栈平衡、局部变量等数据的
  - 数据并不会作为代码
  - DEP：将栈所在的内存置为不可执行
  - 不会执行恶意代码
  - 会导致Crash

DEP

- DEP的全称是“Data Execution Prevention”，是微软随Windows XP SP2和Windows 2003 SP1的发布而引入的一种数据执行保护机制。
  - 2003.9：AMD CPU开始支持NX，即“No eXecute”
  - 随后：Intel CPU开始支持“Execute Disable”（或EDB）或“XD-bit”
  - 2004年8月6日：XP SP2推出， 支持DEP

DEP


上一步
索引(N)
收藏夹(A)
历史(Y)
支持(U)
选项(O)
搜索
设置搜索选项
帮助和支持中心
Windows XP Professional
了解数据执行保护
数据执行保护可帮助保护您的计算机免受病毒和其他安全威胁的破坏。这些病毒和威胁尝试从受保护的内存位置运行（执行）恶意代码来发起攻击，而只有 Windows 和其他程序才应使用这些位置。这种威胁通过接管程序正在使用的一个或多个内存位置来执行破坏操作。之后，它会进行传播，从而破坏其他程序、文件乃至您的电子邮件联系人。
与防火墙或防病毒程序不同，DEP 无法帮助防止有害的程序安装在计算机中，而是对您的程序进行监视，确定它们是否能够安全地使用系统内存。要执行监视操作，DEP 软件既可以独立运行，也可以与兼容微处理器协作，将某些内存位置标记为“不可执行”。如果程序尝试从受保护的内存位置运行代码（无论是否为恶意代码），DEP 均将关闭程序并向您发送通知。
DEP 可以利用软件和硬件支持。要使用 DEP，您的计算机必须运行 Microsoft Windows XP Service Pack 2 (SP2) 或更高版本，或者 Windows Server 2003 Service Pack 1 或更高版本。DEP 软件独立运行时可帮助防御某些类型的恶意代码攻击，但要充分利用 DEP 可以提供的保护功能，您的处理器必须支持“执行保护”功能。执行保护是一种基于硬件的技术，用于将内存位置标记为“不可执行”。如果您的处理器不支持基于硬件的 DEP，则最好将其升级为能够提供执行保护功能的处理器。
再次运行被 DEP 关闭的程序是否安全？
安全，但前提是您要针对该程序启用 DEP。Windows 可继续检测企图从受保护内存位置执行代码的尝试，并能够帮助防止攻击。如果启用 DEP 后程序无法正常运行，您可从软件发行商处获取与 DEP 兼容的程序版本，从而降低安全风险。有关 DEP 关闭程序后应如何操作的详细信息，请单击“相关主题”。
如何确定我的计算机上是否启用了 DEP？
1. 要打开“系统属性”，请依次单击“开始”、“控制面板”、“性能和维护”，然后单击“系统”。
2. 单击“高级”选项卡，之后单击“性能”下的“设置”。

数据执行保护-DEP

- DEP的实现机理是把堆栈的页属性设置为NX
- Windows中的DEP选项
  - Optin：默认仅将DEP保护应用于Windows系统组件和服务，具备NX标记的程序自动保护（个人版默认）
  - Optout：为排除列表程序外的所有程序和服务启用DEP（服务器版默认）
  - AlwaysOn：对所有进程启用DEP 的保护，不能关闭
  - AlwaysOff：对所有进程都禁用DEP，不能启动

思考

- 在实施DEP（数据执行保护）之后如何构造Shellcode?
  - 对策:
    - Ret2Libc
    - ROP （Return - Oriented Programming）
    - JOP
    - COP 等

Ret2Lib

- Return-to-library technique
  - 简称“Ret2Lib”，把返回地址直接指向系统某个已存在的函数（如WinExec）
  - 缺点：
    - 只能执行一些简单的代码，缺乏灵活性，对执行代码的功能有很大的限制。

ROP攻击[Shacham,CCS'07]

- ACM CCS 十年最具影响力的研究-2017
- 重写返回地址构造shellcode链
  - Ret-to-libc
  - 设置假的返回地址和参数
  - 通过假的函数地址调用libc函数
- 无注入代码!

ROP

- ROP的思想是通过在libc等库的代码字节序列中寻找 以ret结尾的指令序列，然后将这些指令序列链接，形成能完成简单操作的指令序列，例如加法运算，称之为gadget，并通过将这些gadget串联来完成一些复杂的操作。
- Gadget集合需要具有图灵完备的表述能力，拥有如下的基本操作：Load/Store、算术和逻辑运算、流程控制（无条件跳转，有条件跳转，系统调用，函数调用等）。
- ROP的关键在于分析libc库中的代码片段获取有效的指令片段，然后利用这些指令片段去构建gadget，再使用这些gadget去构建一组操作（gadgets）进而实施运算和攻击。

ROP攻击[Shacham,CCS'07]

Vulnerable Program:

…
jmp *ebx
…

*ebx 可控

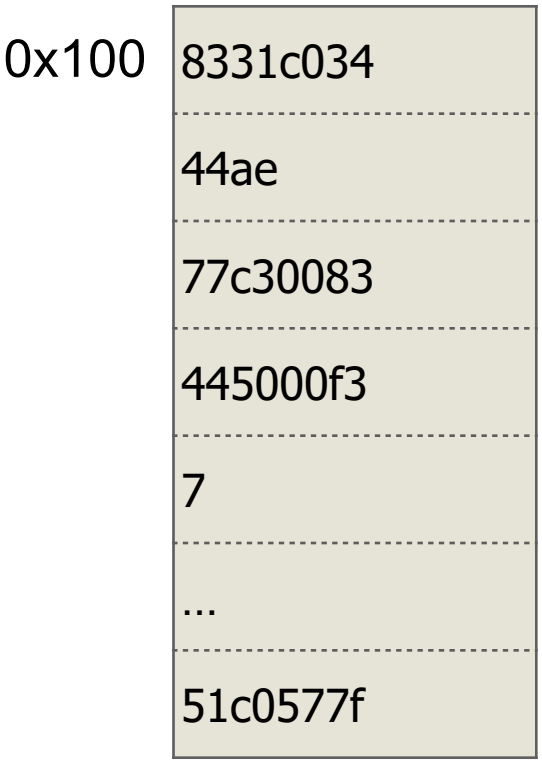
目标shellcode

pop eax
mov edx, eax
pop ecx
`call` VirtualProtect

ROP

Vulnerable Program:

…
jmp *ebx
…



Gadgets

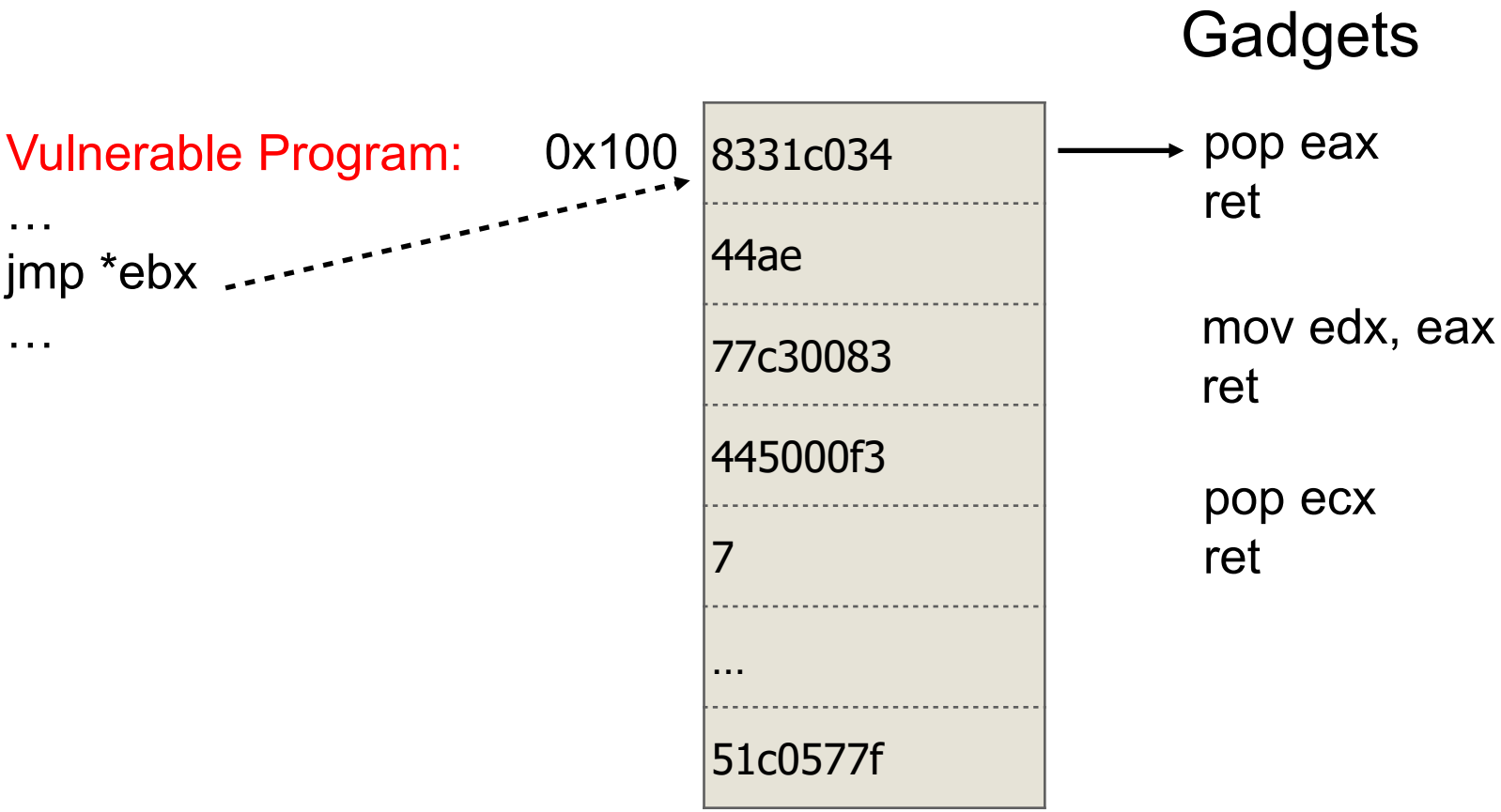
pop eax
ret

mov edx, eax
ret

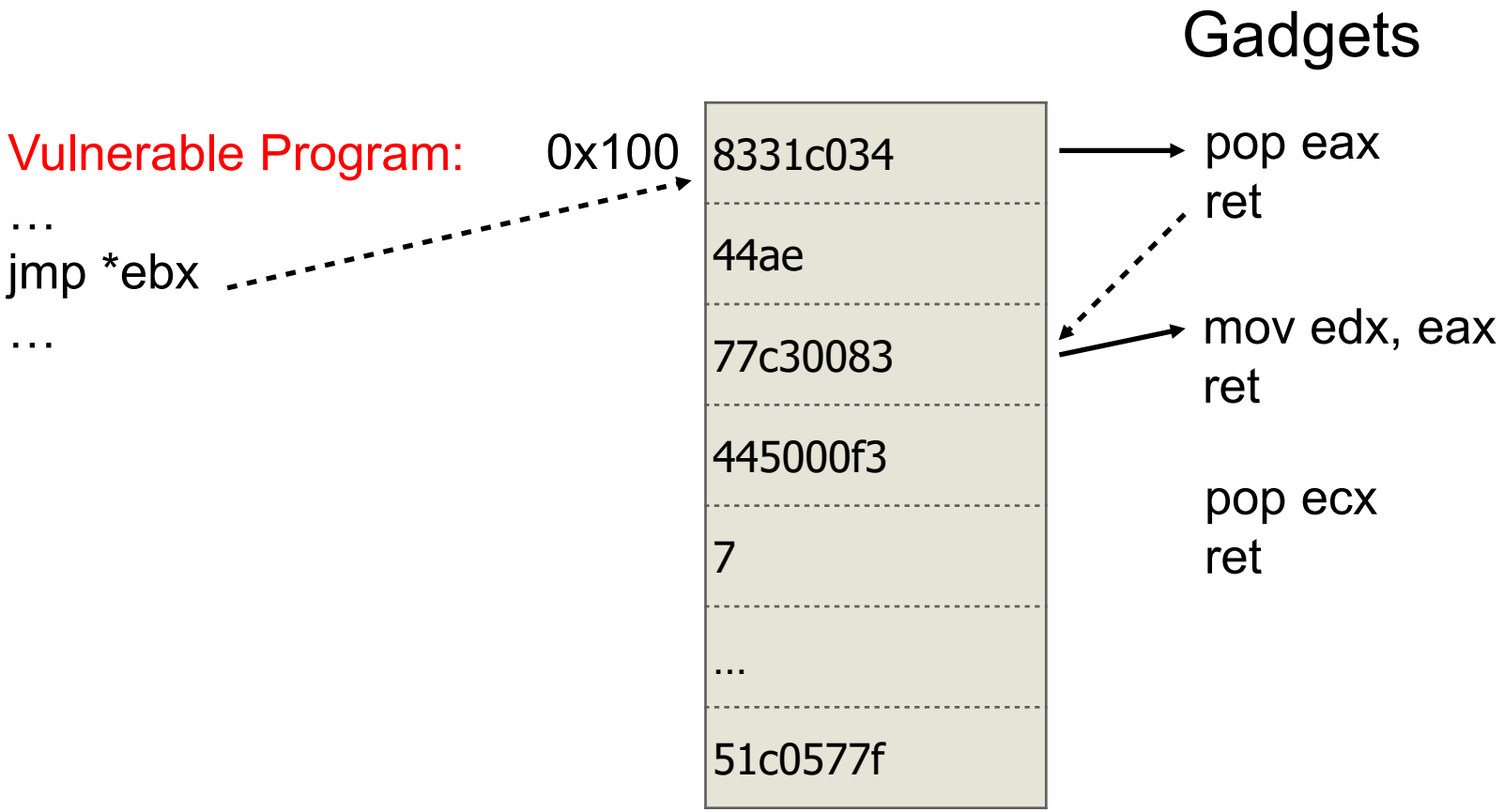
pop ecx
ret

ROP

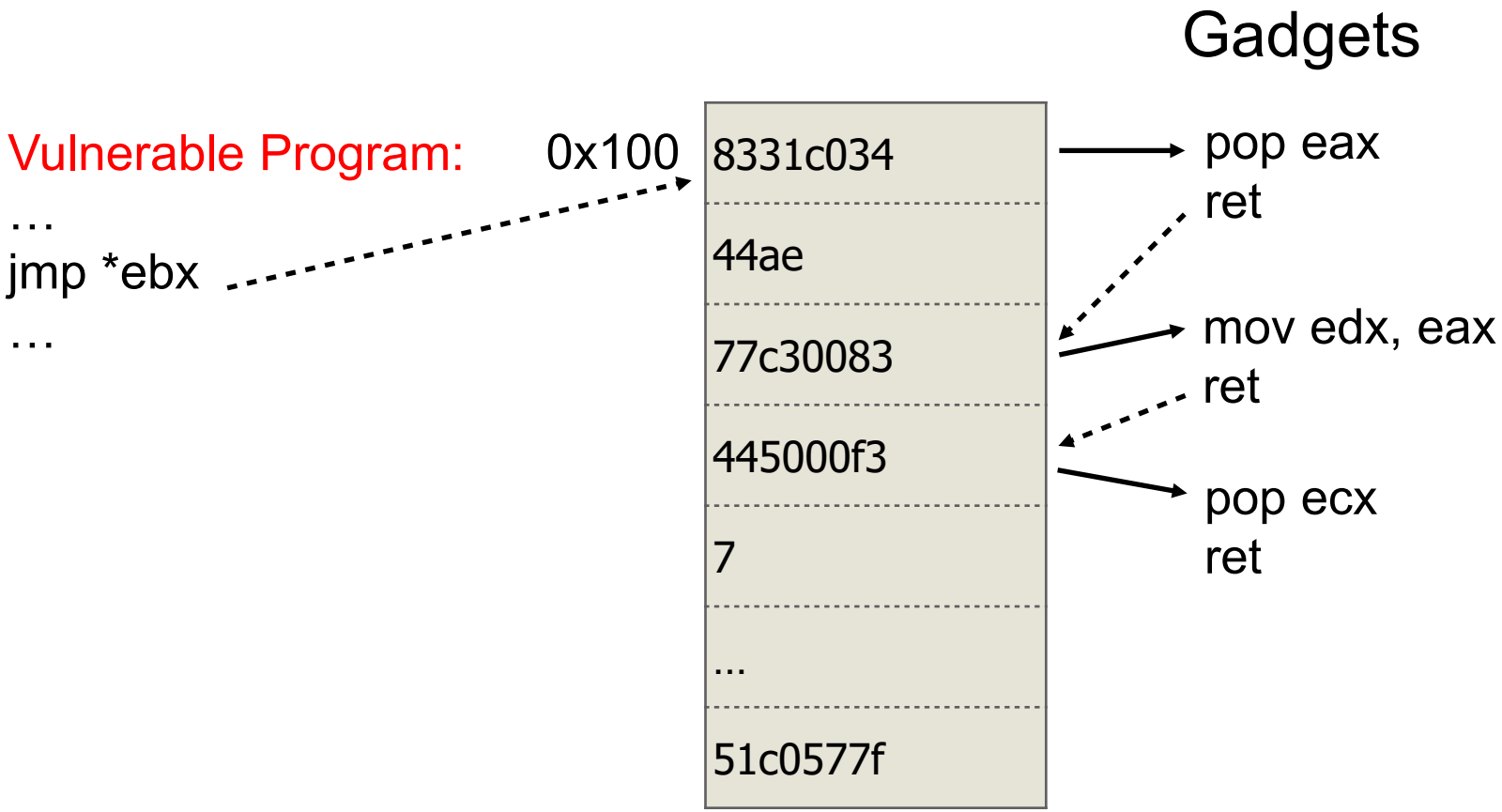
[Shacham,CCS'07]



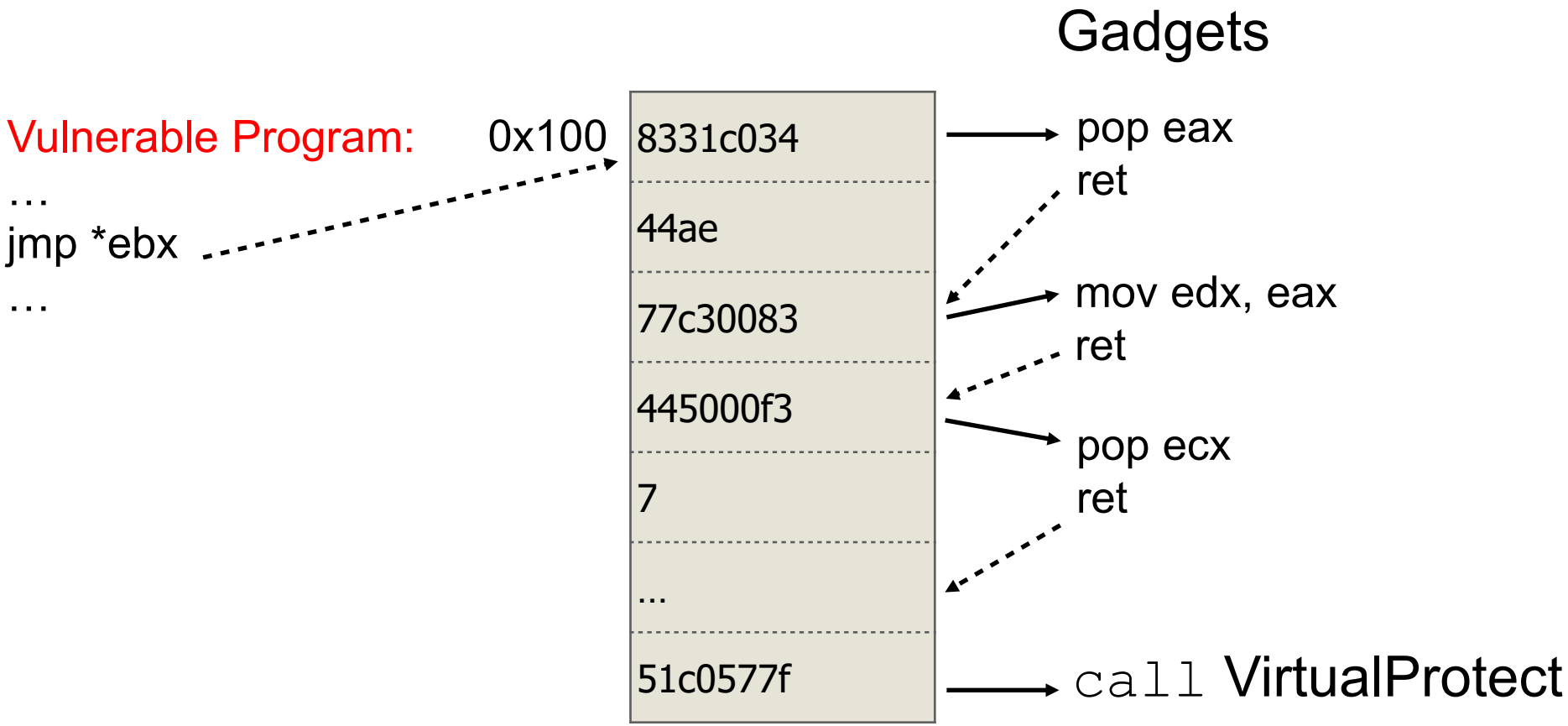
ROP


Gadgets
Vulnerable Program:
0x100
…
jmp *ebx
…
8331c034
44ae
77c30083
445000f3
7
...
51c0577f
pop eax
ret
mov edx, eax
ret
pop ecx
ret

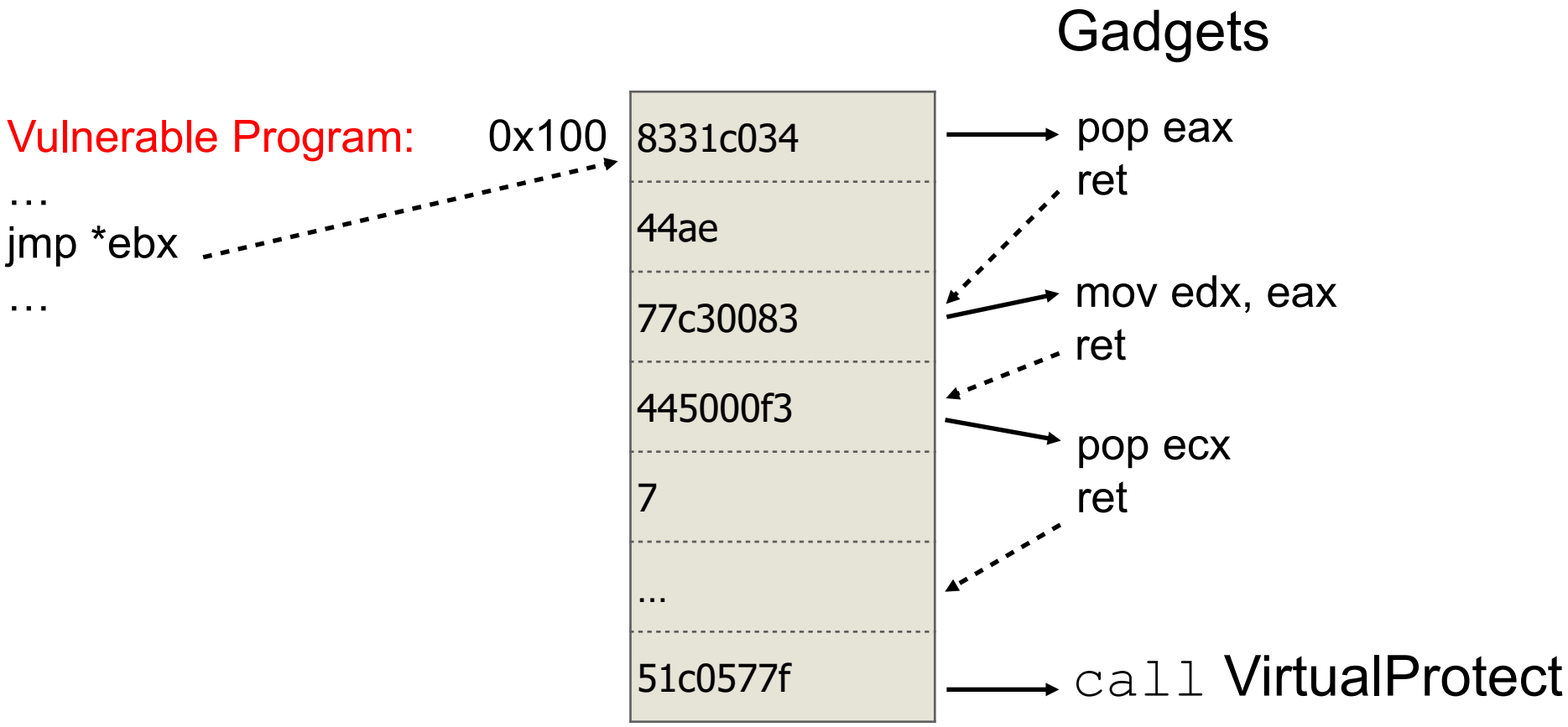
ROP


Gadgets
Vulnerable Program:
0x100
…
jmp *ebx
…
8331c034
44ae
77c30083
445000f3
7
...
51c0577f
pop eax
ret
mov edx, eax
ret
pop ecx
ret

ROP


Gadgets
Vulnerable Program:
0x100
…
jmp *ebx
…
8331c034
44ae
77c30083
445000f3
7
...
51c0577f
pop eax
ret
mov edx, eax
ret
pop ecx
ret
call VirtualProtect

ROP



Gadgets无处不在

gadget自动化构建

- ROP攻击的核心是gadget构建，然后通过gadget来构造自己的POC代码。

- 基本思想如下：
  - 通过程序自动化扫描分析程序库，发现有效的段指令序列，获得短指令序列集合，
  - 分析这些短指令，构建图灵完备的gadget集合，然后用这些gadget集合作为基础指令。
  - 构建一门自定义的语言，写出攻击代码，然后这门语言会翻译为gadget基础指令。
  - 最后得到一长串指令集合。

思考

- 如何检测ROP（COP / JOP）？

DEP评价

性能	• 需硬件支持，无影响 • 否则，据研究表明开销<1%
部署	• 内核支持（所有平台） • 可选模块
兼容性	• 会破坏合法程序 • 即时编译 • 脱壳
安全保障	• 将代码注入到禁止执行页时将永远无法执行 • 代码注入可能是不必要的

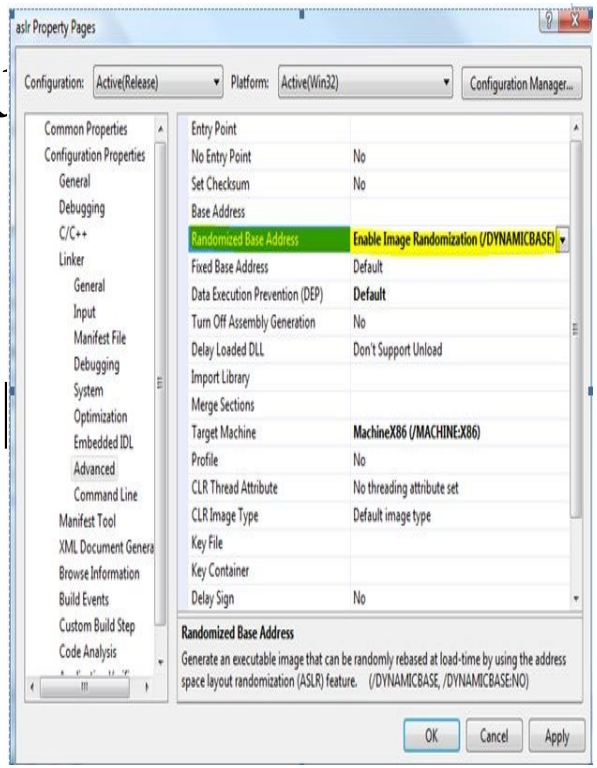
Address Space Layout Randomization (ASLR)

ASLR地址空间布局随机化

- 传统的利用方法需要准确的地址
  - 栈溢出：shellcode位置
  - return-to-libc： 库函数位置
  - ROP: gadget的位置
- 虚拟内存环境下，程序各段的布局是固定的
  - 堆、栈、库等
- 解决办法：将各个区域的地址随机化

地址空间布局随机化-ASLR

- 部署Shellcode
  - 知晓堆或者栈的地址
  - 借用程序自身或者库中的代码（如
- ASLR的思想
  - 栈和堆的基址是加载时随机确定的
  - 程序自身和关联库的基址是加载时



ASLR

- ASLR（Address Space Layout Randomization），地址空间格局的随机化，对主模块、DLL模块、堆、栈的起始地址随机化。

地址空间布局随机化-ASLR

- Windows中的ASLR
  - 从Visual Studio 2005 SP1开始，增加了/dynamicbase链接选项。
  - /dynamicbase选项可以通过Project Property -> Configuration Properties -> Linker -> Advanced -> Randomized Base Address
  - PE结构：IMAGE_DLLCHARACTERISTICS_DYNAMIC_BA

DllCharacteristics：DLL的文件属性，只对DLL文件有效，可以是下面定义中某些的组合：

```cpp
#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040     // DLL can move.
#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY    0x0080     // Code Integrity Image
#define IMAGE_DLLCHARACTERISTICS_NX_COMPAT    0x0100     // Image is NX compatible
```

ASLR示例


```
int main( int argc, char* argv[] )
{
   HMODULE hMod = LoadLibrary( L"Kernel32.dll" );
   char StackBuffer[256];

    void* pvAddress = GetProcAddress(hMod, "LoadLibraryW");
   printf( "Kernel32 loaded at %p\n", hMod );
   printf( "Address of LoadLibrary = %p\n", pvAddress );
   printf( "Address of main = %p\n", main );

   foo();
   printf( "Address of g_GlobalVar = %p\n", &g_GlobalVar );
   printf( "Address of StackBuffer = %p\n", StackBuffer );

   if( hMod )
     FreeLibrary( hMod );

   system("pause");
   return 0;
}
```

操作系统重启前后

武漢大學
WUHAN UNIVERSITY

	重启前	重启后
kernel32 基址	0x776D0000	0x76780000
Loadibrary 函数地址	0x777228D2	0x767D28D2
Main 函数地址	0x00DA1020	0x013B1020
Foo 函数地址	0x00DA1000	0x013B1000
全局变量 g_GlobalVar 的地址	0x00DA336C	0x013B336C
StackBuffer 数组地址	0x0021FC28	0x001BFDC0

ASLR评价

- 开销小，只需要在加载时随机化一次即可
- 开启内核支持
- 无需二次编译
- 对安全应用透明
- 代码注入可能是不必要的

ASLR的绕过

- 暴力猜解
- 并非全部代码都是随机化的
- 内存泄露+ROP链构造
- GOT表劫持

暴力猜解绕过ASLR

- 可行：hacking blind （2014 S&P）
  - 背景：很多服务器程序使用fork等创造工作进程
  - 工作进程和管理进程的地址空间布局一样
  - 工作进程崩溃之后，管理进程立即创建新的工作进程
  - 重启后的新工作进程？

ret2text

- 代码段包含可执行的程序代码
- 但并不会被ASLR随机化除了PIE

- 可劫持控制流至非预定的程序函数

当函数指针可修改时

- 修改函数指针使其指向：
- 程序函数（类似于ret2text）
- 另一个过程链接表(PLT)中的库函数

```
/*please call me!*/
int secret(char *input) { … }

int chk_pwd(char *intput) { … }

int main(int argc, char *argv[]) {
    int (*ptr)(char *input);
    char buf[8];

    ptr = &chk_pwd;
    strncpy(buf, argv[1], 12);
    printf("[] Hello %s!\n", buf);

    (*ptr)(argv[2]);
}
```

相对偏移不变

- 特殊的gadge：
  - pop/pop/…/ret
  - pop eax; call eax
  - pop eax, jump eax
  - ……
-

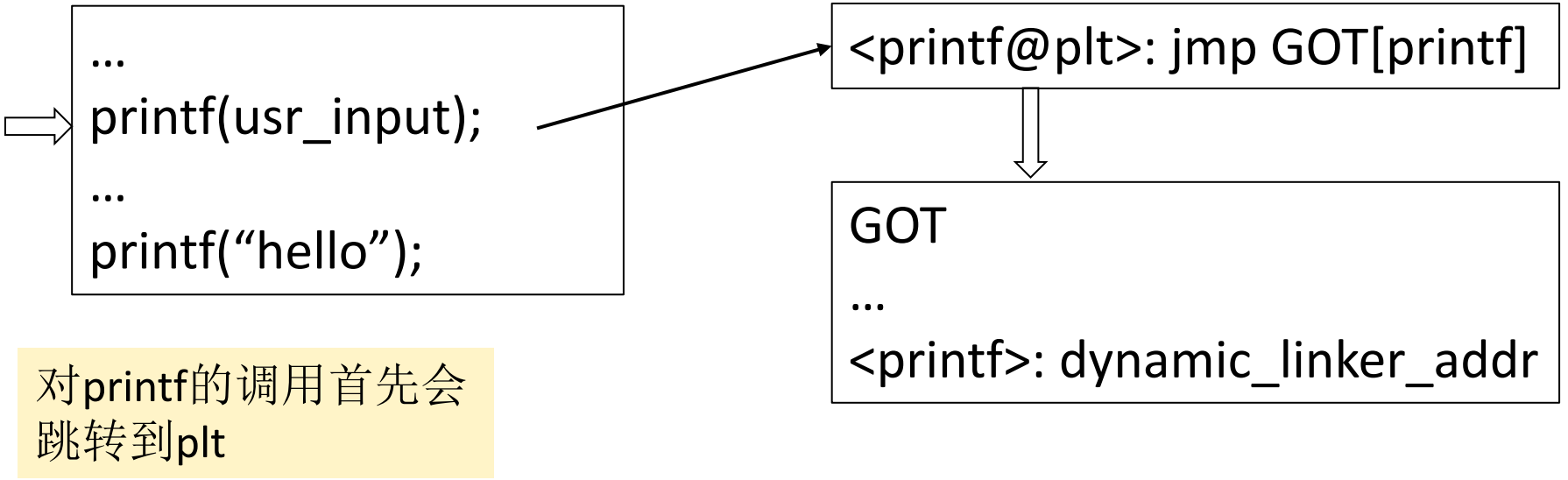
其它非随机部分

- 动态链接库在程序运行时加载，这种链接方式称为lazy binding.

- 两种重要的数据结构
- 全局链接表
- 过程链接表

通常在编译时静态定位

GOT劫持

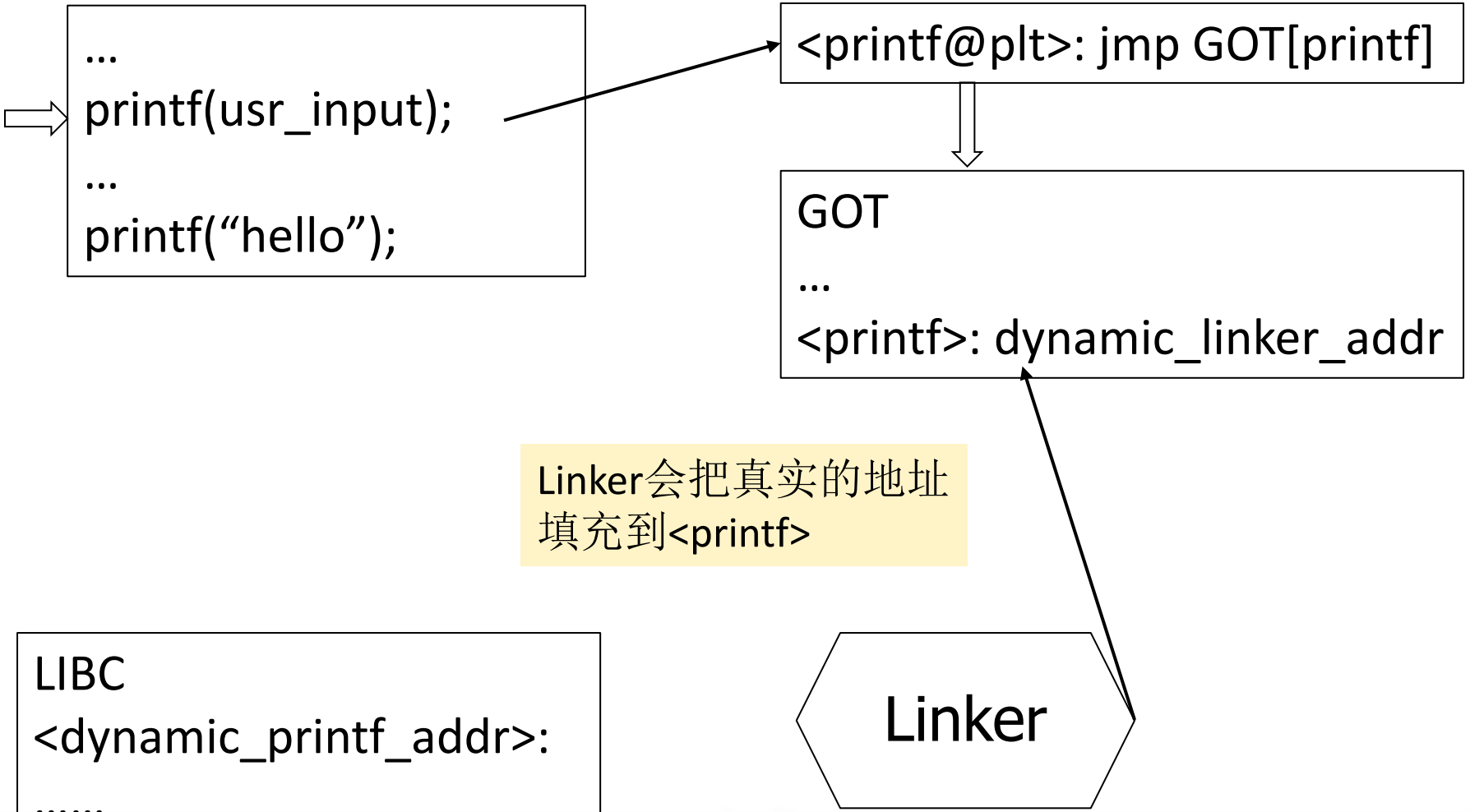


对printf的调用首先会跳转到plt

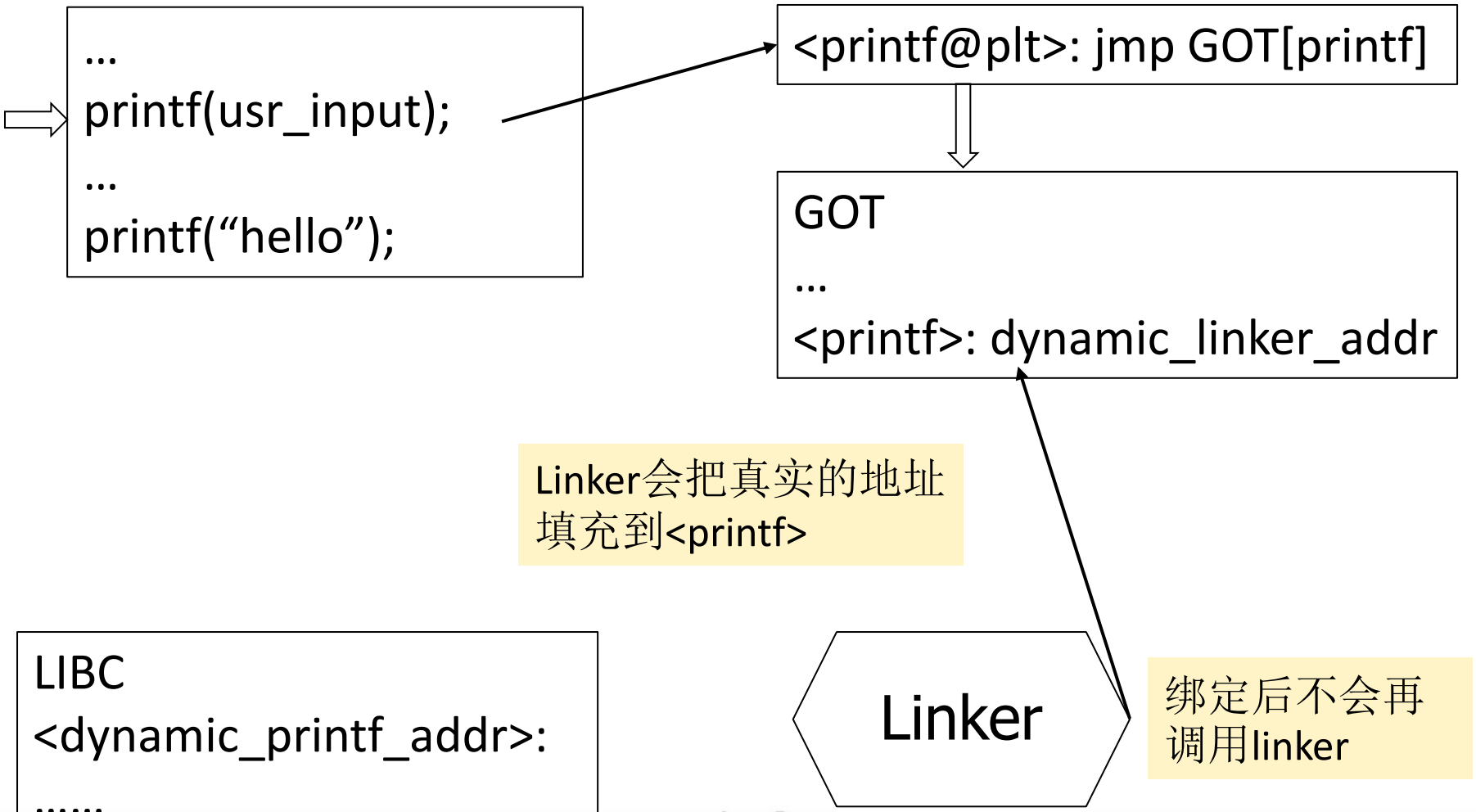
LIBC
<dynamic_printf_addr>:
……

Linker

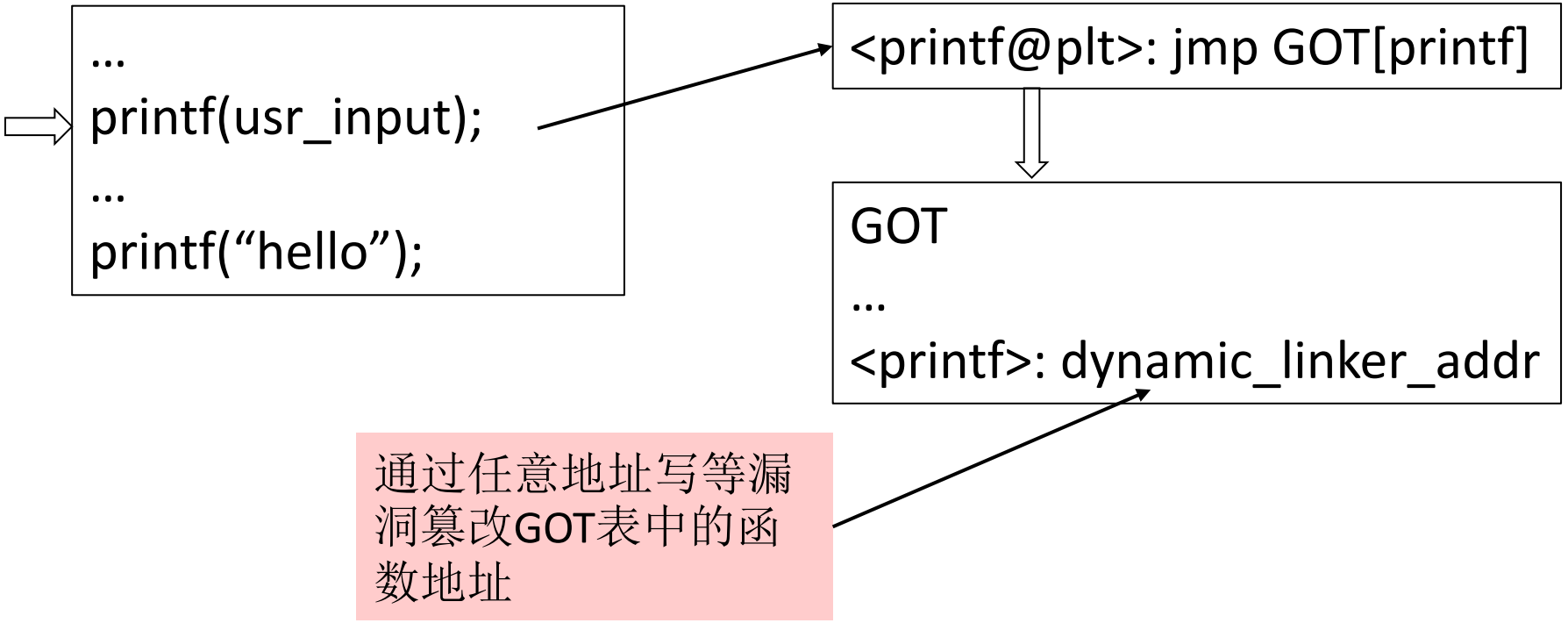
GOT劫持


…
printf(usr_input);
…
printf(“hello”);
<printf@plt>: jmp GOT[printf]
GOT
…
<printf>: dynamic_linker_addr
Linker会把真实的地址填充到<printf>
LIBC
<dynamic_printf_addr>:
……
Linker

GOT劫持


…
printf(usr_input);
…
printf(“hello”);
<printf@plt>: jmp GOT[printf]
GOT
…
<printf>: dynamic_linker_addr
Linker会把真实的地址填充到<printf>
LIBC
<dynamic_printf_addr>:
……
Linker
绑定后不会再调用linker

GOT劫持


…
printf(usr_input);
…
printf(“hello”);
<printf@plt>: jmp GOT[printf]
GOT
…
<printf>: dynamic_linker_addr
通过任意地址写等漏洞篡改GOT表中的函数地址
LIBC
<dynamic_printf_addr>:
……
Linker

JIT-ROP

- 假如程序里含有一个内存泄露的漏洞
  - 利用内存泄露输出库函数中的代码
  - 通过基准地址，找到gadget随机化后的地址
  - 构造ROP链

基于对象伪造的堆区漏洞利用

堆区的数据结构

- 对象
  - 函数指针
  - 堆区对象释放后数据不擦除

- 哪些类型的对象
  - C++对象中虚表及虚函数指针
  - 中间语言的对象
    - 以IE为例，含VBScript和JScript等多个解释器
    - VB和JS的对象都是保存在堆区
    - IE神洞

武漢大學
WUHAN UNIVERSITY

漏洞分析与漏洞检测

漏洞研究

- 漏洞研究
  - 漏洞挖掘
    - 人工代码审计、工具分析挖掘
  - 漏洞分析
    - 漏洞机理、触发条件、漏洞危害
  - 漏洞利用
    - 编制触发漏洞的POC，或者攻击者实施攻击目的的程序（Exploit）
  - 漏洞防御
    - 从软件本身修补、虚拟补丁、防病毒工具升级、IDS升级、防火墙升级

漏洞利用

- 漏洞来源
  - 黑客自己挖掘的漏洞
    - 0-Day VUL
  - 从公开发布的POC或者黑客交换得到的漏洞
    - 0-Day VUL or 1-Day VUL
  - 从已发布的漏洞公告和漏洞补丁获得的漏洞
    - n-Day VUL
- 漏洞利用的条件
  - 用户没有打补丁或者更新安全工具
  - 管理员没有打补丁或者更新安全工具
  - 有漏洞软件的碎片化或多样性

漏洞利用的思路

- 利用目标：
  - 修改内存变量 （邻接变量）
  - 修改代码逻辑（代码的任意跳转）
  - 修改函数的返回地址
  - 修改函数指针(++)[虚函数]
  - 修改异常处理函数指针（SEH,VEH,UEF,TEH）
  - 修改线程同步的函数指针
    - RtlEnterCriticalSection，RtlLeaveCriticalSection
    - ExitProcess将调用

漏洞利用的思路

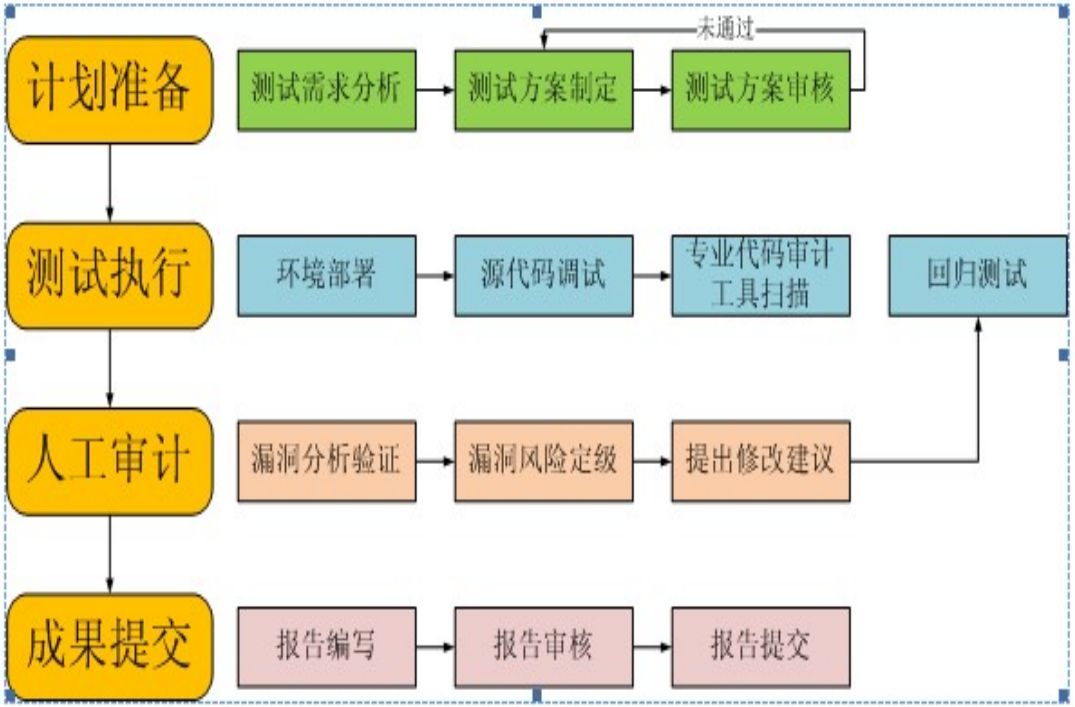
- 利用过程：
  - 定位漏洞点：利用静态分析和动态调试确定漏洞机理，如堆溢出、栈溢出、整数溢出的数据结构，影响的范围
  - 按照利用要求，编写Shellcode
  - 溢出，覆盖代码指针，使得Shellcode获得可执行权

软件漏洞挖掘技术

- 源代码审计
- 动态分析技术
- Fuzzing测试
- 面向二进制程序的逆向分析
- 基于补丁比对的逆向分析

源代码安全审计

- 源代码安全审计
- 词法分析和语法分析
- 构建控制流图、数据流图
- 审计危险函数
  - 字符串操作函数
  - 内存拷贝函数
  - 文件操作
  - 数据库操作
  - 进程操作
  - 安全检查函数



静态分析工具

- 依据CVE(Common Vulnerabilities & Exposures)公共漏洞字典表、OWASP十大Web漏洞，以及设备、软件厂商公布的漏洞库，结合软件设计规范或者编程规范对各种程序语言编写的源代码进行安全审计的工具。
- RIPS-能够检测XSS、SQL注入、文件泄露、本地/远程文件包含、远程命令执行以及更多种类型的漏洞。
- Fortify SCA- 多语言支持

5.4.2 静态分析工具

- Fortify SCA- 多语言支持
  - 数据流引擎：跟踪,记录并分析程序中的数据传递过程所产生的安全问题
  - 语义引擎：分析程序中不安全的函数,方法的使用的安全问题
  - 结构引擎：分析程序上下文环境,结构中的安全问题
  - 控制流引擎：分析程序特定时间,状态下执行操作指令的安全问题
  - 配置引擎：分析项目配置文件中的敏感信息和配置缺失的安全问题
  - 特有的X-Tier™跟踪器：跨跃项目的上下层次,贯穿程序来综合分析问题

静态分析工具

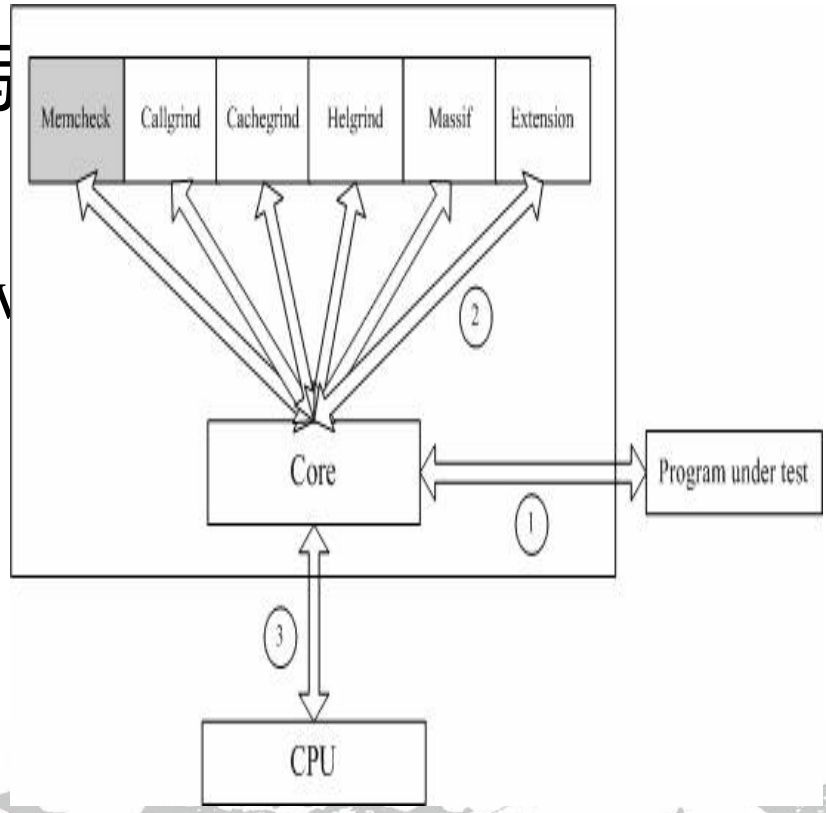
- Findbugs-Java Bug pattern
  - 正确性（Correctness）：这种归类下的问题在某种情况下会导致bug，比如错误的强制类型转换等。
  - 最佳实践反例（Bad practice）：这种类别下的代码违反了公认的最佳实践标准，比如某个类实现了equals方法但未实现hashCode方法等。
  - 多线程正确性（Multithreaded correctness）：关注于同步和多线程问题。
  - 性能（Performance）：潜在的性能问题。
  - 安全（Security）：安全相关。
  - 高危（Dodgy）：FindBugs团队认为该类型下的问题代码导致bug的可能性很高。

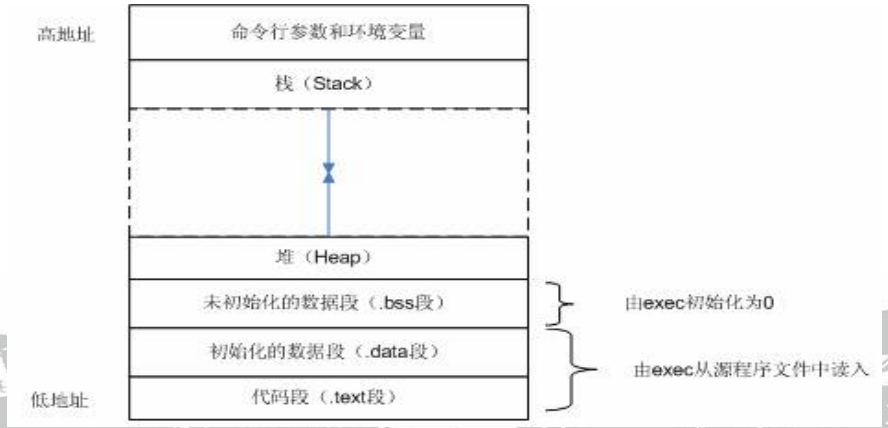
动态分析

- 静态分析的结果不准确，存在误报较多
- 动态分析
  - 收集程序多次执行的运行过程的状态信息，结合输入和输出，检测程序存在的缺陷或漏洞。
- 分析粒度：指令、分支、函数、系统调用
- 分析方法：动态切片、污点传播

动态分析

- MEMcheck：是Valgrind下的内存检查工具
  - 使用未初始化内存
  - 对释放后内存的读/写
  - 对已分配内存块尾部的读/写
  - 内存泄露
  - 不匹配的使用malloc/new/nev
  - 重复释放内存





动态分析

- BitBlaze
  - Vine将汇编语言翻译成一种中间语言（IL）并提供一系列在IL上进行静态分析的核心工具（包括控制流，数据流，优化，符号化执行，最弱前提计算）
  - TEMU进行动态分析，以支持系统全局的细粒度监控和动态二进制仪表。它提供了一系列工具用于提取操作系统级语义，用户自定义的动态污点分析，以及一个用于用户自定义活动的插件接口
  - Rudder利用Vine和TEMU提供的核心功能在二进制代码级进行具体及符号化混合的执行。对于一条指定的程序执行路径，它能给出满足要求的符号化输入参数。

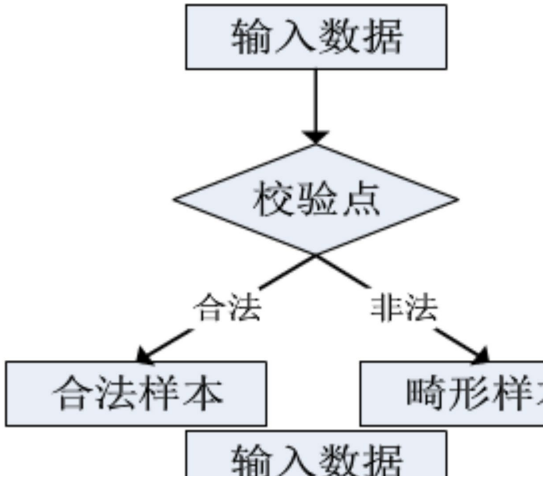
Fuzzing测试

- Fuzzing是一种基于错误注入的黑盒随机测试技术，向目标程序输入随机或者半随机（semi-valid）的测试集，分析程序的执行结果及检测程序状态，发现目标程序中隐藏的各种安全漏洞
- 该方法源于1989年Barton Miller在威斯康星大学提出
- Fuzzer能很好的检测出可能导致程序崩溃的问题，如缓冲区溢出、跨站点脚本，拒绝服务攻击、格式漏洞和SQL injection等

Fuzzing测试

- 难点
  - 输入值在特定小范围引发的异常
  - 几个输入同时作用下引发的异常
  - 需满足特定数据/文件格式后才引发的异常

- SmartFuzzer
  - 输入的格式分割
  - 输入字段的约束
  - 字段之间的约束



Fuzz测试实验

- 阅读AFL的源代码，利用AFL的二进制工作模式，找出crash
  - AFL　http://lcamtuf.coredump.cx/afl/

- 分析报告要求
  - 掌握AFL的二进制工作模式原理
    - 算法描述、工作流程图
  - 能够使用AFL对可执行文件进行测试
    - 针对ubuntu下的常见程序，如PowerDNS, Bind和dnsmasq
    - 测出crash有课程加分

面向二进制程序的逆向分析

- 程序理解
  - 算法的理解学习
  - 代码检查
  - 代码比较
  - 查找恶意代码
  - 查找软件Bug
  - 查找软件漏洞
- 代码优化
  - 平台间的移植
  - 修补Bug
  - 增加新的特性
  - 代码恢复优化

面向二进制程序的逆向分析

- 流程逆向
  - 关键函数（加密、认证）
  - 函数的关联
- 数据格式逆向
  - 格式的分割
  - 类型的推理
  - 字段值溯源
- IDA(Hex-rays)，OD，GDB，WinDBG

基于补丁分析的逆向分析

- 补丁
  - 漏洞
  - 功能优化
- 补丁分析
  - 漏洞机理
  - 功能优化
- 分析方法
  - 基于指令相似性的图形化比较
  - 结构化二进制比较
- BinDiff能够针对x86、MIPS、ARM/AArch64、PowerPC等架构进行二进制文件的对比。