软件安全基础

用户关心正确性，安全阻止不希望出现的行为。

不希望的行为：机密性（窃取信息）、完整性修改信息或功能）、可用性（拒绝访问）。

CIA

机密性：防止未授权的用户访问数据——不能看，采取加密或权限类保护措施。

完整性：防止未授权的修改数据——不能改。严格的访问控制、身份认证。

可用性：保证经过授权的客户及时准确访问数据——一直用。

缺陷与漏洞

漏洞利用：利用一个与安全相关的软件缺陷实现预期之外的行为。

软件缺陷：软件行为不正确，无法满足设计需求。

漏洞：安全方面的缺陷，使得系统或其应用数据的机密性、完整性、可用性受到威胁。

内存安全

32位CPU：一次处理32bit数据。

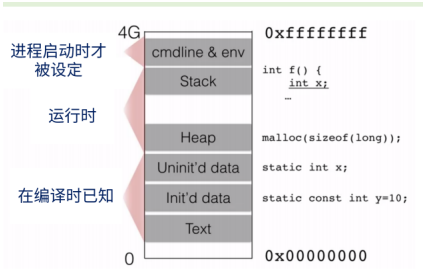
32位编译模式：一个地址占用4字节内存。

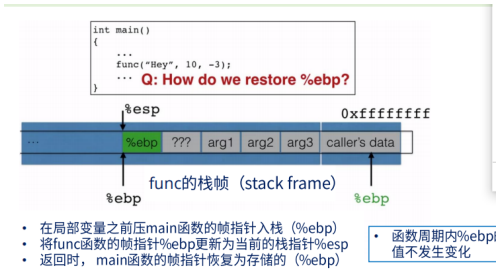
栈

存储局部变量、函数参数、返回地址。编译器自动分配和释放。栈顶由栈指针寄存器%esp标记，存放最后一个压入栈顶项的地址。%ebp记录帧指针。

堆

程序员手动分配的内存。





- 设%eip为离%ebp 4 bytes的地址
- 在函数调用前，压下一个要执行的指令地址（%eip）入栈

- 调用函数：
  - 1. 将参数压入栈（反向）
  - 2. 压入返回地址，即返回后你想要运行的指令的地址
  - 3. 跳到要执行的函数的地址
- 被调用函数：
  - 4. 将旧帧指针压入栈（%ebp）
  - 5. 将帧指针 %ebp 设置为当前栈顶的位置（%esp）
  - 6. 将局部变量压入栈
- 函数返回：
  - 7. 重置之前的栈帧：%ebp = （%ebp）
  - 8. 回到返回地址：%eip = 4（%ebp）

代码注入

将代码载入内存：机器代码指令，不包含全零字节。

让其运行：设置eip。

其他攻击

堆溢出、整数溢出。

缓冲区过读。

悬空指针（地址已被收回free只释放内存未收回指针）

内存安全特性

内存安全程序：标准方法创建指针、访问“属于”该指针的内存。

- 格式化字符串攻击违反了空间安全准则
- 对printf的调用引用了非法指针

```
char *buf = "%d %d %d\n";
printf(buf);
```

  - 将栈视为由其提供的参数的数量和类型定义的缓冲区
  - 额外的格式说明符构造超出此缓冲区末尾的指针，并诱使printf进行了引用（读取）

- 本质上是一种缓冲区溢出

空间安全：保证是合法区域。

时序安全：保证区域仍然有效。

悬空指针：已释放、未初始化。

内存安全的编程语言：Java、C#、Python。

大部分语言是内存安全的，C、C++不是，编译器添加代码来检查。

避免漏洞利用

栈警惕标志

在栈返回地址存储位置前放置一个整型值（在装入程序前确定），栈由低到高覆盖栈空间，返回前检查标志是否被篡改。

攻击者难以猜测和利用。

终止符（0x00）、**随机数**（不可被读取发现）、**随机XOR**（随机数和栈的控制信息、返回地址异或）。

Stack Guard等均主要基于该技术。

局限：只对控制信息和返回地址进行保护，溢出一些局部变量可会造成攻击，不能防御堆。

栈数据不可执行（解决eip指向攻击代码）

使栈的内存段不可执行。

局限：有些攻击不需要执行栈上的代码，如Return-to-Libc。

Return-to-Libc不需要执行栈，而是执行libc(存放标准c语言库函数)中的一个函数例如system（）函数，通过eip跳转到库中。

ASLR地址空间布局随机化

解决eip指向攻击代码，找到返回地址。

安全代码实现

Rule：输入校验

```
len = MIN(len,strlen(buf));

    if(i<0 || i>9)
      return '?';
    return convert[i];
```

Principle：信任最小化

Rule：安全string函数

Rule：不忘记NUL终止符

- 使用安全的string操作函数将捕获此错误

```
char str[3];
strlcpy(str,"bye",3);    // blocked
int x = strlen(str);     // returns 2
```

Rule：理解指针运算、防御悬空指针。

Rule：使用goto避免重复、遗漏代码。

Rule：使用安全函数库。

Web安全

Outline

Web1.0 带状态Web Web2.0

防御：验证输入，然后信任。

与Web服务器通讯

URL

因特网上标准的资源的地址。

组成：协议+域名+路径+(参数)。

HTTP

超文本传输协议。用于分布式、协作式和超媒体信息系统的应用层协议。

提供一种发布和接收HTML页面的方法。

Web的核心。

定义浏览器如何从Web服务器请求Web页面，以及服务器如何把Web页面传给客户。

HTTPS

安全超文本传输协议。

由HTTP进行通信，SSL/TLS加密数据包。

提供对网站服务器的身份认证，保护数据的隐私与完整性。

Web通讯的基本结构

请求：URL+header+GET/POST

响应：状态代码+header+数据+cookie

HTTP常用状态码：

- 状态代码的第一个数字代表当前响应的类型：
  - 1xx 消息——请求已被服务器接收，继续处理
  - 2xx 成功——请求已成功被服务器接收、理解、并接受
  - 3xx 重定向——需要后续操作才能完成这一请求
  - 4xx 请求错误——请求含有词法错误或者无法被执行
  - 5xx 服务器错误——服务器在处理某个正确请求时发生错误

有状态Web

维持状态：将状态发给客户，客户端在后续响应返回状态。

隐藏字段：

- 服务器维护可信状态（客户端维护其余部分）
  - 服务器存储中间状态
  - 发送一个 capability 给访问该状态（state）的客户
  - 客户端在后续响应中引用该 capability

缺点：繁琐，必须在回访时重新开始。

Cookie：

网站为辨别用户身份存储在用户本地中断上的数据（加密）。

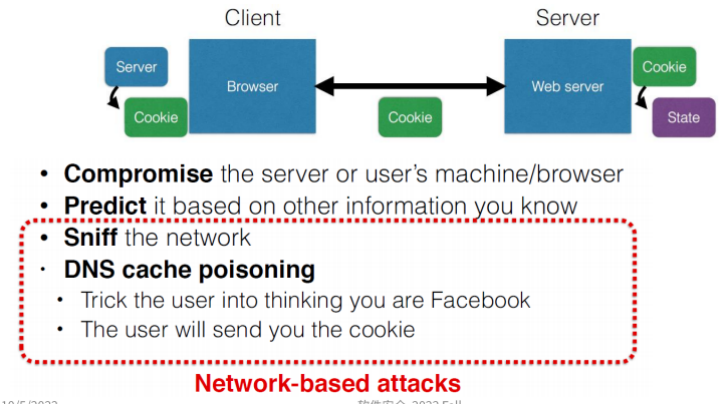
作用：会话标识符，个性化。

- 服务器维护可信状态
  - 服务器使用 Cookie 来索引状态
  - 将 Cookie 发送到客户端，客户端保存 Cookie
  - 客户端后续查询返回 Cookie 给同一服务器

Cookie：key-value对，Set-Cookie，保存在客户端。

扩展：浏览器指纹。

窃取Cookie：



保护：

cookie随机选择、足够长，需要提供独立的关联信息。会话超时机制。

非保护（基于IP）：误报、漏报。

动态Web

网页可表示为JavaScript编写的程序。

脚本嵌入在返回的网页，脚本由浏览器执行。

同源策略

两个页面**协议、端口、域名**均相同。

限制了同源加载的文档、脚本与其他源的资源交互。（隔离恶意文件）

SQL注入

SQL语言

SQL，结构化查询语言。

```
SELECT：SELECT Age FROM Users WHERE Name='xx';
UPDATE：UPDATE Users SET email='xxx' WHERE Age=32;
INSERT INTO：INSERT INTO Users Values('xx','xx',xx,...);
DROP：DROP TABLE Users;
```

SQL注入

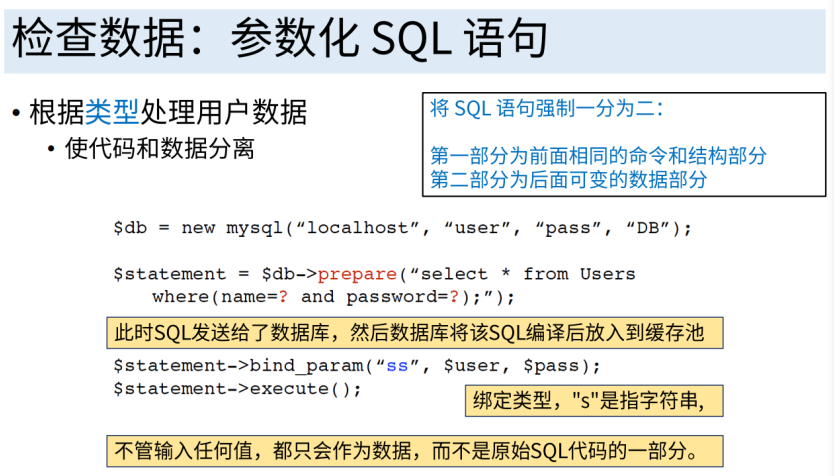
SQL注入，把SQL命令插入到Web表单提交或输入域名或页面请求的输入参数，达到欺骗服务器执行恶意SQL命令的目的。

根本原因：代码数据界限模糊。

对策：

1.输入校验（检查形式、数据消毒）。

2.消毒：黑名单（删除或替换敏感字符）、白名单（检查用户输入的安全、默认失效安全safer to reject than to fix）。



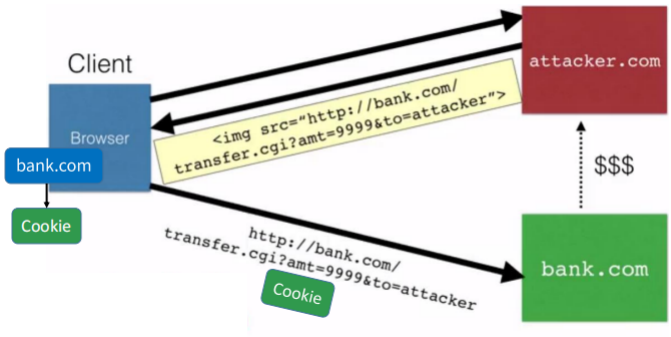
深度防御（缓解）：最小权限、加密敏感数据。

跨站请求伪造CSRF

同站请求：网页给它所在网站发HTTP请求。

跨站请求：网页的来源和请求的去处不是同一网站。

总结：利用网站对用户浏览器的信任。



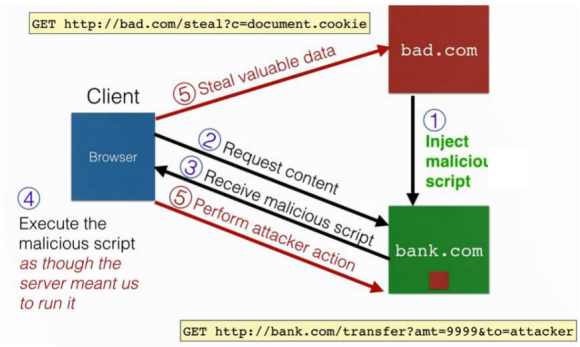
保护：验证HTTP Referer字段（安全性依赖于第三方，可篡改，涉及隐私）。请求地址中添加token（不存放于Cookie）。

跨站脚本攻击XSS

attacker提供恶意脚本诱使用户相信脚本源是目标网址。

用户浏览器运行脚本，具有与服务器提供的脚本的相同访问权限。

存储型XSS

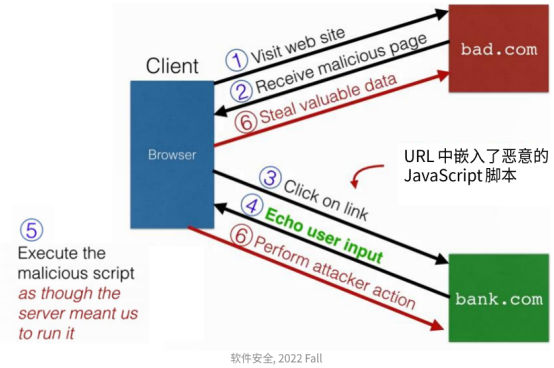


服务器无法确保上传页面内容不含嵌入脚本。

目标：支持JavaScript浏览器的用户，访问有漏洞服务器受用户影响下的内容页面。

工具：服务器上存留内容的能力。

反射型XSS



目标：支持JavaScript浏览器的用户，访问有漏洞Web服务，服务输出可能嵌入外部脚本。

工具：用户点击特制的URL。

服务器无法确保其输出不包含外部嵌入式脚本。

防御

过滤、数据消毒：删除...（引入js的方法很多）

白名单：允许全部不如进行限制。

- XSS 攻击利用客户端浏览器对从合法网站发送的数据的信任
  - 因此攻击者试图控制网站发送到客户端浏览器的内容
- CSRF 攻击利用合法网站对客户端浏览器发送的数据的信任
  - 因此攻击者试图控制客户端浏览器发送到网站的内容

验证然后信任！

白名单优于黑名单-默认安全

检查优于数据消毒-减少信任

竞争条件

在多线程、进程同时发生交互的环境出现。

漏洞窗口。

事件次序异常造成对资源的竞争使程序无法正常运行。

竞争条件在高度受控的环境不易出现。

“原子化”：关联代码和数据执行时一个整体，操作执行不会有其它事发生。（加锁原语）

TOCTOU

检查时间与使用时间。

利用条件：

1.攻击者必须可以访问本地机器。

2.有竞争条件的程序必须以root的EUID运行。

3.程序必须在竞争条件存在时间内保持EUID。

否则攻击者不能获得root权限。

防御

原子化、反复检查使用竞争条件、粘滞位、最小权限。

粘滞目录：只有目录内文件所有者或root可以删除移动文件。

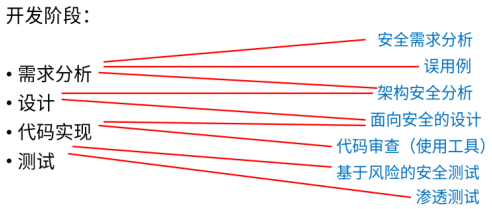
尽量选择单线程而避免多线程。

其他竞争条件

支付系统（额外数据库跟踪）、脏牛漏洞。

安全软件开发

四个常见阶段：



软件：具有可塑性、易改，利于核心功能但对安全有害。

硬件：速度快但难改，具有安全优势。

威胁模型：假定对手攻击能力。模型中做出的假设都是对手可以利用的潜在漏洞。

安全需求：CIA。

支撑机制：身份认证、授权、审计。

身份认证：确认操作者身份，将行动与身份联系起来。

授权：被授权者何时可以执行某项操作。

审计：保留足够信息以确定违规或不当行为。

误用例：描述系统不该做什么（更改账户汇率）。

安全设计原则

预防：完全消除软件缺陷。

缓解：减少未知缺陷造成的伤害。

检测：识别并理解攻击，还原伤害。

尽可能简单

使用默认失效安全：配置和使用选择会影响安全性，默认选择应该是安全的。

不要指望专家级用户：考虑用户思维模式和能力，默认提供安全性而不是繁琐操作后获得安全性。

谨慎信任

减少需求信任：不作出不必要的假设、使用更好的设计和流程。

最小权限：只授予为执行某操作而必须的最小访问权限，只分配访问所需的最小时间。

输入验证是一种最小权限验证。

尽可能限制敏感信息的流量（例如pdf只能看不能下载）。

隔离：在特定区域或沙箱隔离特定系统组件，无法进行交互来降低权限。断开数据库与internet连接，linux一种沙盒seccomp沙盒（只允许四种系统调用命令）

分割：隔离的同时，系统分割为小单元降低损害。

深度防御

使用多种防御策略管理风险。

监控和跟踪

记录相关操作信息。

漏洞分级与报告

CVE

负责任批漏

CVSS漏洞评级：基础（可利用指标和影响指标）、时间、环境。

程序分析

软件测试导引

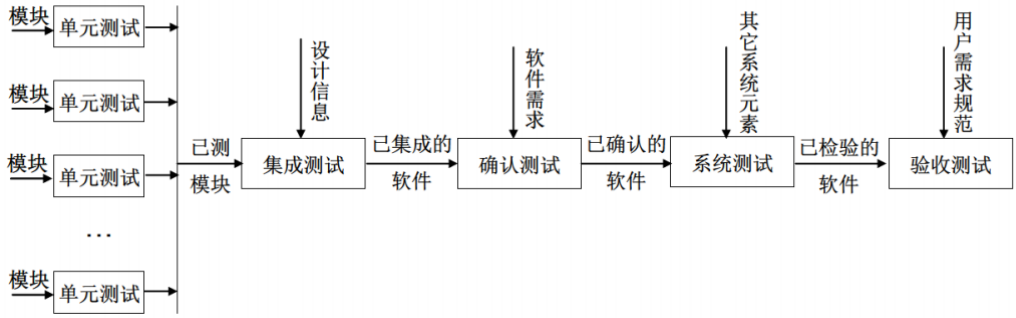
软件测试：需求分析、设计规格说明和编码的最终复审，随着软件规模和复杂性增加日趋重要。

只有通过软件测试才能发现软件缺陷，只有发现缺陷才能将其从软件产品中清除。

软件测试的正面性：为了验证软件产品能否正常工作。

软件测试的反面性：测试是为了证明程序有错。

软件测试过程



单元测试：针对程序最小单元——模块/组件进行白盒测试，从内部结构出发设计测试用例，检查每个单元已实现功能与定义功能是否一致。

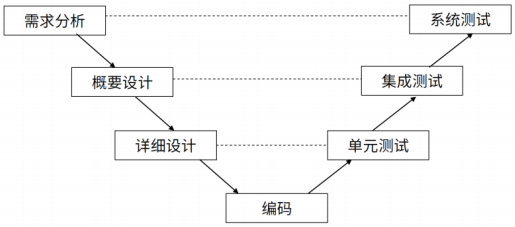
> **白盒测试**是按照程序内部的结构测试程序，通过测试来检测产品内部动作是否按照设计规格说明书的规定正常进行，检验程序中的每条通路是否都能按预定要求正确工作。白盒测试一般用来分析程序的内部结构，对测试者而言是透明的，测试者可以看到被测程序源代码，并分析其内部结构。

集成测试：将模块组装起来进行测试，主要为了发现与接口有关的模块之间的问题。

功能/确认测试：从用户角度进行功能验证，基于产品说明书对功能测试。

系统测试：将软件放在实际运行环境下进行安全、强度、性能等测试，开发人员和组织不能参与。

验收测试：进一步确保软件准备就绪能执行既定功能。



软件测试环境：硬件(设备、配置)+软件(OS)+网络(局域网or互联网，传输带宽等)+数据准备(大量真实的正确与错误数据或者模拟)+测试工具(已有、购买或自行开发，根据需求)。

模拟真实环境，不要安装与测试无关软件，环境与开发环境独立。

静态、动态分析

程序测试：确保程序在输入集上正确运行，出现故障有助于修复，但昂贵、难以覆盖所有代码路径。

代码审计：源代码分析，说服他人源代码是正确的。人类一次可以看出多次错误，所以会优于单次运行，但人力昂贵。

自动化程序分析技术：静态&动态（是否需要运行代码）

静态分析：检查代码或运行自动化方法来查找错误。覆盖率更高，只能分析有限属性，可能漏报误报，运行非常耗时。

- 分析设计权衡
  - 精确度：仔细模拟程序行为，以最大限度地减少误报
  - 可扩展性：成功分析大型程序
  - 可理解性：错误报告应该是具有可操作性的

动态分析：测试条件下运行代码查看可能的问题(比如GDB、Fuzzing、黑盒测试)。

数据流污点分析

获取相关数据沿着程序执行路径流动的信息分析技术（对象执行路径上数据流动和可能取值）。

污点分析：跟踪并分析污点信息在程序中的流动。

许多攻击就是因为信任了未经验证的输入（格式化字符串攻击）

无污点标记数据流：对所有可能输入证明在预期无污点数据的地方永远不会使用污点标记数据（需要推断数据流）。

分析途径：视为类型推断。

1.为缺失的限定符创建名称。

2.对程序每个语句，在可能解上生成约束。

3.求解约束生成解。

增加流敏感

考虑语句执行顺序：允许一个变量每个指定有不同限定符。

路径敏感：精确度限制可扩展性。

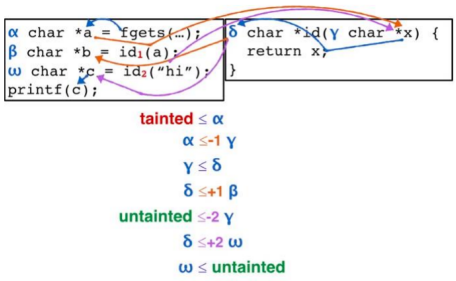
上下文敏感

命名参数和返回值：

调用函数数据→被调函数参数→被调函数结果→调用函数返回值

两次调用同一函数时再次产生误报（检测出的污点流实际是不可能的路径）。

增加上下文敏感：可以区分调用点，将调用与返回相匹配(±i)。



符号执行

会分析得到让特定代码区域执行的输入，过程中使用**符号值**而非具体值作为输入，在目标代码处用约束求解器得到触发目标代码的具体值。

断言不为真程序会终止允许并报错。

每个符号执行路径代表具体满足路径的运行集，可以覆盖更多的程序执行空间。

符号执行是计算密集型，每一个分支语句都可能是指数级的路径增长。计算机速度慢（处理能力不强）和内存小。

基本的符号执行

符号变量：读取输入时引入α、β，出现bug可以恢复程序重现bug输入。

符号表达式：制作修改语言解释器以能够进行符号运算。

符号执行通用化了测试，通过静态分析生成涵盖不同程序路径的测试。

线式执行：

Concrete Memory	Symbolic Memory
x ↦ 5	x ↦ α
y ↦ 10	y ↦ 5+α
z ↦ 17	z ↦ 12+α
a ↦ {0,0,0,0}	a ↦ {0,0,0,0}
Overrun!	*Possible overrun!*

路径条件：

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<10)
5      y = 5;
6  } else y = 0;
```

- 当α> 5时，达到第3行
- 当α> 5且α<10时，到达第5行
- 当α≤5时，达到第6行

路径与断言：

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 if(z < 0)
5   abort()
6 if(z >= 4);
7   abort();
8 a[z] = 1;
```

π = true
π = true
π = true
π = true
π = 12+α<0
π = ¬(12+α<0)
π = ¬(12+α<0) ∧ 12+α≥4
π = ¬(12+α<0) ∧ ¬(12+α≥4)

abort()函数终止程序

防止了越界访问

Fuzzing

一种随机测试：补充功能测试，直接测试功能（以及缺少错误功能）、正常测试可以作为模糊测试的起点。

目标：确保某些不好的事情不会发生，不管怎样崩溃、抛出异常、不终止所有这些都可以成为安全的基础漏洞。

一种随机测试，将自动或半自动生成的随机数据输入程序中并监视程序异常，确保崩溃、异常、未终止等（安全漏洞的基础）不会发生（监视程序异常、报告程序错误）。

模糊测试分类：

黑盒：对程序一无所知，容易使用，但只能探索浅层状态。

基于文法：可以更深入状态空间。

白盒：至少部分根据被测试程序代码生成新输入，容易使用，但计算昂贵。

模糊测试输入：对合法输入进行转变，从头开始，组合（这块翻译不出来摆了）

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
  - Generate initial input, mutate^N, generate new inputs, …
  - Generate mutations according to grammar

渗透测试

模拟黑客的手法对网络或主机进行攻击测试，积极尝试查找可利用的漏洞来评估安全性。

通常不是库或不完整代码段。

可应用于不同粒度：程序（进程），应用（进程通讯）、应用程序组成的网络。

通常由单独小组执行，与开发人员分开，利用自己的工具搜索利用漏洞，通常会给予成员从无法访问到完全访问的各种权限。

优点：产生真实漏洞的证据才更利于修复，测试证明渗透是确定和可重复的（不是假设、适于整个组件不是代码段、没有误报）。

缺点：未渗透成功不是安全的证据，系统更改就需要重新测试代价昂贵。

身份认证

计算机（网络）确认操作者身份，决定用户是否具有资源的访问使用权限。

·What you know

·what you have

·Who you are

口令认证

基于主机

依赖MAC地址、IP地址、cookies、处理器ID。

物理令牌

钥匙、信用卡。

为每个用户提供输入设备、遗失被盗、复制。

生物认证

签名、指纹、虹膜。

需要物理输入设备、输入机制安全性（篡改认证设备获得数据）、生物行为可变、唯一但不保密、侵犯隐私。

深度防御：多因素认证

口令+验证码

口令存储

保存并使用口令散列：口令→认证代理散列→比较哈希。

彩虹表：加密散列函数逆计算表。

加盐：口令任意固定位插入特定字符串，使其不同于原始口令的散列。对于每个口令，盐值独一无二。

加盐的意义：防止复制口令在口令文件中可见。防止彩虹表批量攻击（口令长度增加→猜测难度增加）。攻击者无法发现一个用户是否使用相同口令。

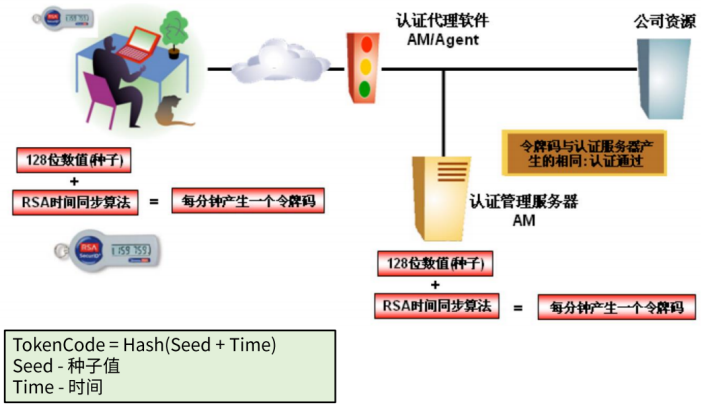
一次性口令

基于服务器和客户端的时间同步（安全令牌精确时钟与服务器同步）。

使用算法根据之前的密码生成新秘密。

新密码基于挑战或计数器（服务器选择随机数）。

双因素口令=PIN+令牌码



哈希链

实用安全

可用安全

以人为中心设计和搭建的安全系统。

人机交互：系统与用户交互。

可用性评估：有专家评估、用户实验、问卷访谈等。

安全技术人员不关心用户需要，我们应当考虑用户如何与系统进行交互

安全增加了障碍。不可用的安全是容易，可用的安全是困难的。

策略

尽可能简单。

默认失效安全：配置或使用选择会影响系统安全性如密钥长度、口令；默认选择保证安全。

不指望专家用户：考虑用户思维模式和能力对安全的影响，如简单界面、避免频繁决策、帮助用户探索后果、自然的选择是安全的。

案例

SSL警告、Android权限、验证码、身份认证（可用性vs安全性）。

软件保护技术

客户端安全：**可用性**、**性能**折中。

版权保护

许可证密钥

许可证文件：根据用户的特定信息产生与用户相关的许可证。

传送给用户经过数字签名处理的文件→应用程序中内嵌签名公钥进行验证。

包含安装时间的许可证以限制许可时间。

将机器信息纳入许可证范围，提高破解难度。

其他方案：查询手册输入指令、加密狗(一种硬件，程序运行时会检查加密狗是否存在)、远程执行（向服务器申请验证，运行在服务器上，需要一直在线）

防篡改

增加修改软件的难度但不让软件崩溃。

重复多次检查，让攻击者重复多次破解，这显著增加了代码开销。

数据校验

文件校验：程序启动时计算文件的校验值与先前对比判断是否篡改。（检查签名）

内存校验：计算可能遭篡改的代码区块，对内存区域与计算校验和，运行时动态比较（会暴露关键代码位置<混淆或进行大量校验>）。

应对滥用

当检测到无权限使用或篡改时，让程序出错停止。为防止找到处理攻击的代码，让检测攻击和处理攻击代码放的远一些。

保留程序原有的错误，仅当没有篡改异常时修复处理原有错误。

防止程序被调试/动态分析

检测到调试器运行，程序终止。

父进程检测（windows原理：以程序被正常启动，父进程应该是资源管理器、命令行或系统服务，若都不是可认为被调试了）。

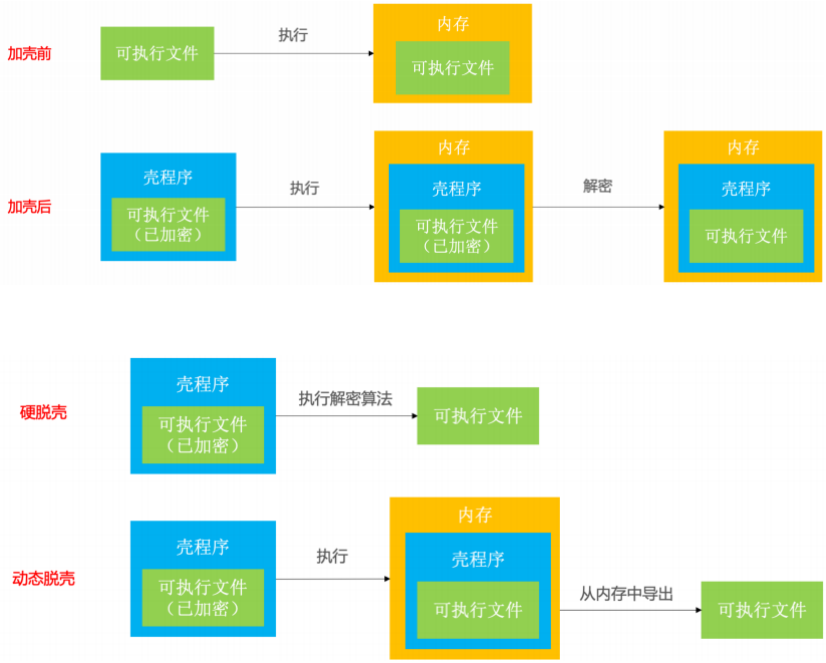
调试器逐行运行耗时多，通过计算运行时间判断。

系统痕迹检测

对抗动态分析：虚拟机检测。通常mac前几位标识了提供商，检测出虚拟机。分辨真实用户与自动化测试工具输入。

软件加壳:

在应用程序中加入一段保护层代码使源代码失去本来面目。运行时先执行外壳程序，外壳程序将原程序在内存中解开。



代码混淆

防御反汇编。

将程序代码转换成功能等价但难以阅读理解的形式。

代码混淆器可以混淆源代码或中间代码。

保留清晰的源程序树，便于维护。

基本方法

增加从不执行的代码。

添加总是评价为真或错的条件。

变量名更改。

独特的数据编码（字符串看不懂）。

程序控制流。

代码混淆技术评价

效能：人有多难理解。

抗逆：对自动化去混淆工具的抵抗。

花销：运行时间空间的开销，对性能的影响，文件大小的增加。

各种方法提高软件安全性但不可能完全保护知识产权，使破解的代价超出预计的合理的。

随机数

数字序列在统计上随机、不可由已知推算后续。

伪随机数生成器

返回值由前一个返回值决定。

知道"种子"，就可算出所有返回值。

假设攻击者知道PRNG工作原理，深度防御。

加密型PRNG：给予足够熵的加密算法。满足所有传统统计型PRNG目标，并提供安全性。

统计型PRNG：对任意统计测试产生看似随机的数据序列。

防御密码分析攻击&防御内部状态的攻击。

PRNG统计测试：高质量PRNG的输出和真随机数应该是不可区分的

熵收集和处理

硬件设备：熵更高，来自于自然的随机过程（较昂贵）。提供熵的大小取决于源的品质（熵通常保守取）。

软件方案：利用软件源的随机性，无法预测的外部输入（键盘鼠标流量，深度防御思想采用多个源）。

熵的估计：没有统一度量指标。

需要生成用于加密的随机数，确保收集熵足够大!

现实中伪随机数系统

Linux字符设备random：一系列接口收集系统环境噪声加入熵池

/dev/random适于质量要求高，熵池不足会阻塞。

/dev/urandom：非阻塞，会重复利用熵池，输出熵会小于。

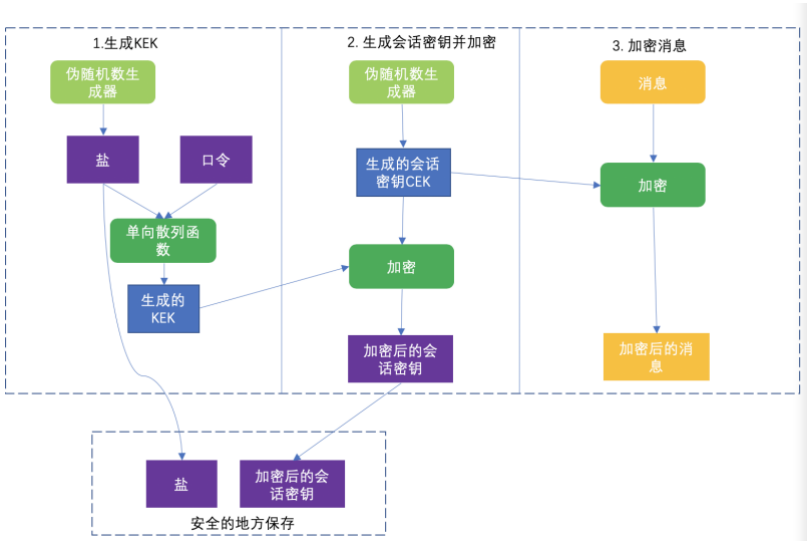
密码技术应用

一般建议

不创建密码、使用公开密码、避免自己设计协议、不使用相同密钥（流密码）、检查数据完整性（MAC共享密钥）、不使用硬编码的密钥。

- 伪随机数发生器（PRNG）的弱随机种子。
- 弱加密算法：随着计算能力的增长和设计漏洞的发现，一些算法已经被证明是不安全的，如DES 和RC4。
- 密钥长度不足：由于攻击技术的改进，短密钥容易受到暴力穷举攻击。
- 分组加密算法的ECB（Electronic Codebook）模式：在ECB 模式下，当明文用相同的密钥加密两次，密文总是相同的，即可识别明文问题。
- 分组加密算法的CBC（Cipher-Block Chaining）模式的弱初始化向量（IV，Initialization Vector）：初始化向量必须是随机的，且永远不会被重用。（选择明文攻击）

PBE：用盐+口令生成KEK，保护会话密钥CEK，安全地方保存盐+加密后的CEK。



加盐：防御字典/彩虹表。

口令：增加牢靠性。

通过拉伸即多次迭代撒捏函数改良PBE。

误用：迭代次数小于1000，可预测盐，硬编码的口令。

哈希函数

在随机数生成中隐藏统计模式、资源计量。

一次一密

安全分发密钥的困难、与消息等长密钥的开销。

一次一密没有实用性，但衍生出了流密码使用伪随机数生成器。

---复习时间太紧一部分参考了前人的总结---