RSA 签名故障分析

Stefano / 2018-08-21 21:01:52 / 浏览数 2513 技术文章 技术文章 顶(0) 踩(0)

翻译自：https://blog.trailofbits.com/2018/08/14/fault-analysis-on-rsa-signing/

前言

今年春天和夏天的时候，我在Trail of
Bits实习，研究了针对RSA签名的故障攻击建模。我查阅了使用中国剩余定理（CRT）的RSA签名优化和会泄露私钥的诱发性计算错误。
我分析了低级别的故障攻击，而不是数学上下文。在分析了一个toy program、一个mbed TLS的RSA实现后，我确定了在翻转时会在内存中泄露私钥的比特位。

使用RSA-CRT的签名过程

通常来说，RSA签名会使用这个算法： $s = m^d (mod\ n)$
其中，s代表签名，m代表明文信息，d代表私有指数，n代表公钥。这个算法是有效的，但是如果是为了安全起见，使用的数值增加到所要求的的大小之后，计算将会花费相

给定私有指数d，我们计算两个值，dp=d (mod (p-1))和dq=d (mod (q-1))。

然后我们使用这两个值分别计算两个部分签名；第一个部分签名 $s_1 = m^{dp} (mod\ p)$ 第二个部分签名 $s_2 = m^{dq} (mod\ q)$ p (mod q)和q (mod p)的逆都是可以计算出来的，两个部分签名结合到一起形成一个最终签名s，即
$$s = (s_1 * q * qInv) + (s_2 * p * pInv)\ (mod\ n)$$

故障攻击

当两个部分签名中的一个（我们假设是s2，使用q计算生成）不正确时，问题就会出现。

最终签名由两个部分签名合并而成。我们可以通过比较原始信息和 $s^e (mod\ n)$ 来验证签名是否正确，其中e是公有指数。然而，在有故障的签名中，
$s^e (mod\ p)$ 依然和m相等，但是 $s^e (mod\ q)$ 则不相等。

此时，p是 $s^e - m$ 的一个因子，而q不是。
因为p同时也是n的一个因子，所以攻击者可以通过计算n 和 $s^e - m$ 的最大公因子来得到p，n直接除以p就可以得到q，所以攻击者就知道了两个密钥。

使Toy Program产生故障

我开始用C写了一个toy
program，使用中国剩余定理进行RSA签名。这个程序不包含填充和检查，使用标准的RSA签署相当小的数字。我使用调试器手动修改其中一个部分签名，最后产生了一个故

用Manticore翻转比特位

我使用Binary
Ninja来查看我的程序的反汇编并确定我感兴趣的数据的内存位置。当我试图解出私钥时，我能知道在哪里查看。然后我安装并学习了如何使用Manticore，这是由Trail
of Bits开发的二进制分析工具，我将用它来进行故障攻击。

我写了一个Manticore脚本，它将迭代每个连续的内存字节，通过翻转字节中的比特位来改变指令，并执行RSA签名。对于每一次没有导致崩溃或超时的执行，我尝试从输出

```python
def bitflip(cpu, memory_location):
    location = cpu.read_int(memory_location, 8)
    flipped = location ^ 1
    cpu.write_int(memory_location, flipped, 8, force=True)

def solve_private_keys(e, s, m, n):
    p = gcd(pow(s, e)-m,n)
    q = n//p
    private_keys = [p, q]
    return private_keys

def decrypt(p, q, e, s, m, n, encrypted):
    totn = (p-1)*(q-1)
    d = modinv(e,totn)
    if not(d == None) and not(n == 0):
        decrypted = pow(encrypted,d,n)
        return decrypted

def check_private_keys(correct_message, message):
    if correct_message == message:
        return "yes"

for i in range(0x400e00, 0x400eda):
...
    m = Manticore(target)
    @m.hook(0x400c50)
    def hook_bitflip(state):
        cpu = state.cpu
        ru.bitflip(cpu, i)

    @m.hook(0x400eda)
    def hook_getdata(state):
        # collect data using cpu.read_int()
    m.run(timeout = 60)

    if len(partial_signatures) == 0:
        i_data["Partial Signatures"] = "crash"
    else:
        i_data["Partial Signatures"] = partial_signatures
        private_keys = ru.solve_private_keys(e, s, m, n)
        i_data["Private Keys"] = private_keys
        decrypted = ru.decrypt(p, q, e, s, m, n, encrypted_message)
        i_data["Decrypted Message"] = decrypted
        check = ru.check_private_keys(correct_message, decrypted)
        i_data["Correct Private Keys?"] = check
    all_data.append(i_data)

# write to CSV file
```


图1.在toy program中寻找导致错误的地址的部分代码

结果

我总共测试了938个位翻转，发现其中45个，或者说4.8%，成功的生成了正确的私钥。接近55%的位翻转会导致崩溃或超时，意味着程序无法产生签名。大约有31%的位翻

Memory Location	Partial Signatures	Private Keys	Decrypted Message	Correct?
0x400d5f	crash			
0x400d60	[7, 9]	[23, 37]	26	yes
0x400d61	[7, 1]	[23, 37]	26	yes
0x400d62	[7, 1]	[23, 37]	26	yes
0x400d63	crash			
0x400d64	[7, 25]	[23, 37]	26	yes
0x400d65	crash			
0x400d66	[1, 26]	[1, 851]		
0x400d67	crash			
0x400d68	[7, 28]	[851, 1]		
0x400d69	crash			
0x400d6a	[0, 28]	[37, 23]	26	yes

图2. 分析代码的输出

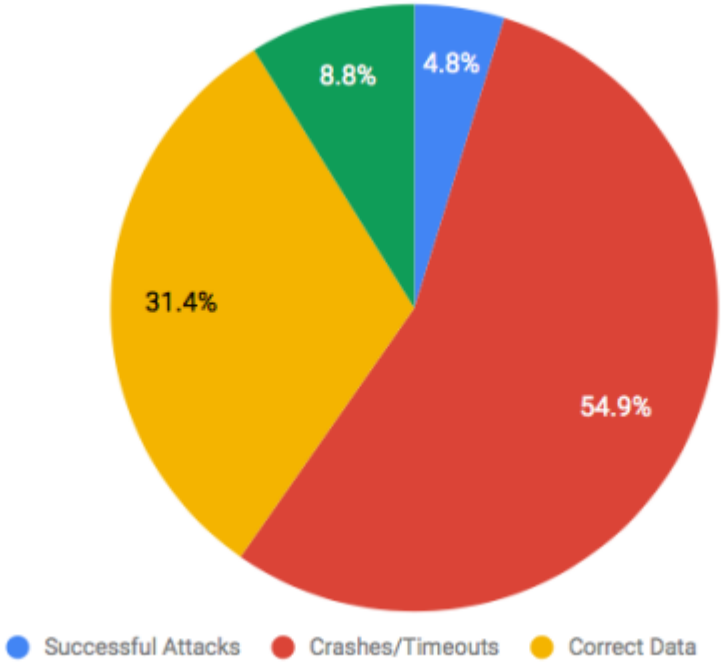


图3. toy program的位翻转结果

这种自动化为挖掘类似的漏洞提供了巨大的速度提升，你只需简单的向Manticore描述一下漏洞，就会得到一个全面的漏洞利用方法表。如果能引入一些不严密的错误（例如

使mbed TLS产生故障

在有了我的toy program的翻转结果文件之后，我找了一个真实的加密库来破解。我选择了mbed
TLS，它主要应用在嵌入式系统上。因其比我写的程序复杂得多，我花了很长一段时间研究mbed TLS的源码，以便在编译前弄懂RSA的签名过程，并使用Binary
Ninja查看了二进制文件的反汇编。

mbed TLS和我的toy program之间的一个关键不通点是使用mbed
TLS的签名会被■■。我试图建模的故障攻击仅适用于确定性填充，给定的消息将始终产生相同的填充值，而不是使用概率性的方案。尽管mbed
TLS已经可以实现许多种不同的填充方案，我另外还查看了使用PKCS#1 v1.5的RSA签名，PKCS#1
v1.5是一种用更复杂的随机PSS填充方案的确定性替代方案。我再次使用调试器来定位目标数据，当我知道我要读取的内存位置时，我使其中一个部分签名产生错误，并生成

然而，我很快意识到，有一些运行时检查可以阻止我进行故障类的攻击。特别是在两次检查失败后，程序会停止执行，输出一个错误信息，不产生签名。我使用调试器跳过检

通过故障签名和所有的公钥数据，我可以复制在我的toy program中成功提取出私钥的过程。

自动化攻击

就像我在toy
program中做的一样，我尝试自动化故障攻击并识别出会泄露私钥的位翻转。为了加快进程，我写了一个GDB脚本，而没有使用Manticore。我找到了一些可以跳过两个检查
program相同的过程中，我在指定的内存地址中额外地翻转了一位。然后我使用Python循环遍历内存中的每一个字节，调用此脚本，尝试提取私钥，同样通过尝试解密另一

```
def solve_private_keys(e, s, m, n):
    p = gcd(pow(s, e)-m,n)
    q = n//p
    private_keys = [p, q]
    return private_keys

def decrypt(p, q, e, s, m, n, encrypted):
    totn = (p-1)*(q-1)
    d = modinv(e,totn)
    if not(d == None) and not(n == 0):
        decrypted = pow(encrypted, d, n)
        return decrypted

# read public keys and collect data for message to check against

for i in range (0x40d27b, 0x40d41b):
    try:
        subprocess.call(["./gdb_flip.sh", str(i)], timeout=60)
        m = mu.read_memory(0x6f08a0, 0x6f09a0)
        i_data = {}
        i_data['Memory Address'] = hex(i)
        s = mu.read_file(sigfile)
        if not(s == "failed"):
            partials = mu.get_partial_sigs()
            i_data['Partial Signatures'] = partials
            os.remove("gdb.txt")
            pkeys = mu.solve_private_keys(e, s, m, n)
            i_data['Private Keys'] = pkeys
            os.remove(sigfile)
            decrypted = mu.decrypt(pkeys, e, n, encrypted)
            i_data['Correct?'] = mu.check_private_keys(correct, decrypted)
        else:
            i_data['Private Keys'] = "failed"
    except subprocess.TimeoutExpired:
        i_data['Private Keys'] = "failed"
        continue
    all_data.append(i_data)

# write data to CSV file
```

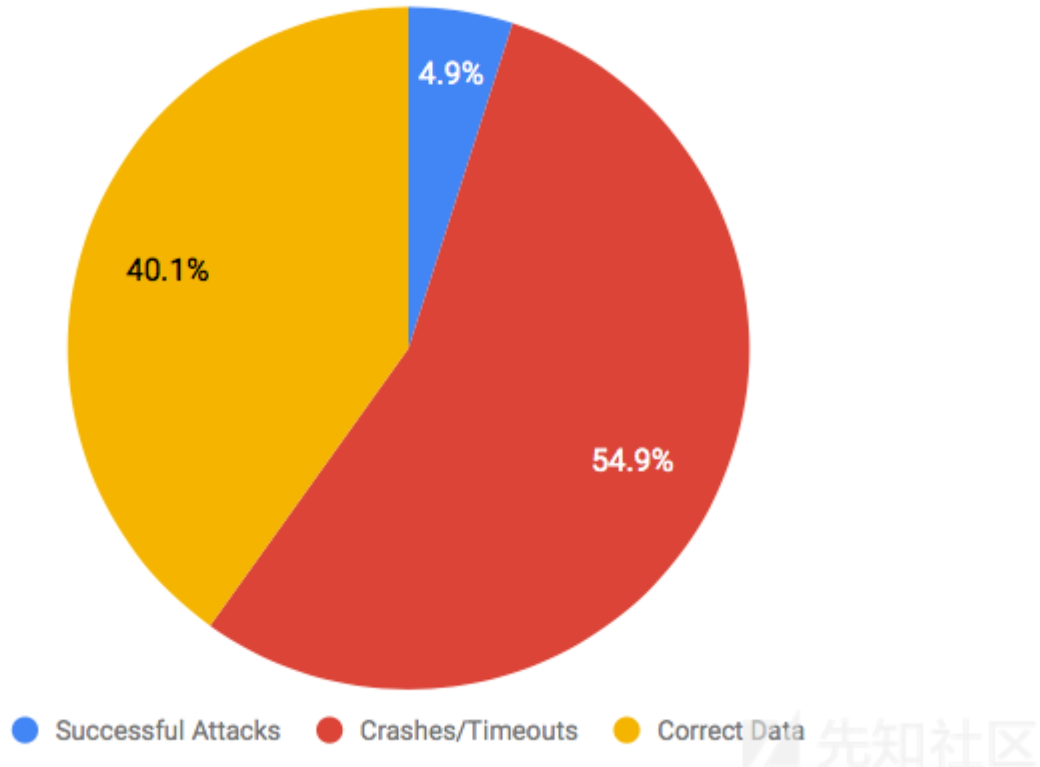

图4. 在mbed TLS中寻找导致错误的地址的部分代码

```
gdb -batch programs/pkey/rsa_sign \
    -ex 'set args "example.txt"' \
    -ex 'set logging on' \
    -ex 'break main' \
    -ex 'break *0x40d27b' \
    -ex 'break *0x40d30d' \
    -ex 'run' \
    -ex 'set *'$address' ^= 0x1' \
    -ex 'set *0x40b7f4 ^= 0x1' \
    -ex 'set *0x40d4f6 ^= 0x10' \
    -ex 'c' \
    -ex 'x /64x 0x6f08a0' \
    -ex 'c' \
    -ex 'x /32x 0x6f0280' \
    -ex 'x /32x 0x6f0310' \
    -ex 'c'
```


图5. 从Python调用的用于在mbed TLS中引发故障的GDB脚本部分代码

结果

我测试了566位翻转，都在mbed
TLS实现签名部分的代码中。结果确保检查通过的两位翻转，发现其中28个——接近5%——泄露了私钥。大约55%未能产生签名。



mbed TLS的位翻转结果

事实上这种分析适用于真实程序是很令人兴奋的，但不幸的是，在我能在"真实的世界"中测试它之前，我已经把夏天的时间花光了。尽管如此，输入真实TLS代码和全面描述

结论

我喜欢在Trail of
Bits工作。我对密码学有了更深的了解，并熟悉了安全工程师使用的一些工具。这是一次美妙的体验，我很高兴能在新的一年开始时将我学到的所有知识应用到卡内基梅隆大

上一篇：【2018年 网鼎杯CTF 第一场... 下一篇：【2018年 网鼎杯CTF 第一场...

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板