CSAW 2014: xorcise challenge

Last weekend our CTF team participated in CSAW. One of the challenges was particularly interesting, and this blog post gives a (somewhat) detailed overview on how to solve it. The source code and binary executable were given. This write-up is also available on the awesome write-ups repository.

Length checking vulnerabilities: Part One

By inspecting the source code we see the binary is a service which listens on port 24001. New clients are handled in `int process_connection(int sockfd)`. This function reads one single packet, which is expected to be in the following format:

```
struct cipher_data {
    /** Header: Unencrypted and unauthenticated */
    uint8_t length; /** Length of the bytes array */
    uint8_t key[8];
    /** Payload: Encrypted and (partly) authenticated */
    uint8_t bytes[128];
};
```

As the comments indicate, the header is sent unencrypted. In particular this header includes an 8-byte key (which is chosen by the client). Roughly speaking the `bytes` array is XORed with the key in blocks of 8 bytes. The function doing this decryption is `decipher(data, output)`, which will be investigated in more detail later on (because it contains a vulnerability). The `length` field the header contains the actual size of the `bytes` array. This must be smaller or equal to 128. An attempt is made to assure the given length is valid:

```
cipher_data encrypted;
ssize_t bytes_read = recv(sockfd, (uint8_t *)&encrypted, sizeof(encrypted), 0);
if (encrypted.length > bytes_read)
    return -1;
```

However this check is flawed. **Variable `bytes_read` is the length of the packet including the header**, while the program will treat the `length` field as the size of the `bytes` array. This means that as an attacker we can force the length to be bigger than 128.

The second vulnerability is in `decipher(data, output)`. This function decrypts the `bytes` array using the key. Somewhat simplified we have the following code:

```
uint32_t decipher(cipher_data *data, uint8_t *output)
{
    uint8_t buf[MAX_BLOCKS * BLOCK_SIZE];
    uint32_t loop, block_index;

    memcpy(buf, data->bytes, sizeof(buf));

    if ((data->length / BLOCK_SIZE) > MAX_BLOCKS)
        data->length = BLOCK_SIZE * MAX_BLOCKS;

    // Block-decryption loop
```

```
    for (loop = 0; loop < data->length; loop += 8)
        for (block_index = 0; block_index < 8; ++block_index)
            buf[loop+block_index] ^= (0x8F ^ data->key[block_index]);

    memcpy(output, buf, sizeof(buf));
}
```

Note that `data->bytes` is copied to a local buffer, and this local buffer is then processed. The first if-test is an attempt to assure that `data->length` is not bigger than the local buffer. However this checked is flawed because in `(data->length / BLOCK_SIZE)` **the intermediate result will be rounded down**. In particular, the following value will pass the length check:

```
data->length = BLOCK_SIZE * MAX_BLOCKS + (BLOCK_SIZE - 1) = 135
```

And due to the first length check vulnerability, we know that `data->length` can indeed contain such values. The consequence of both vulnerabilities means that we can force the block-decryption loop to **decrypt an extra block of 8-bytes**. Since the local buffer is too small for this, we are capable of modifying local variables placed after the buffer. In particular we can overwrite `loop` and `block_index`.

Length checking vulnerabilities: Part Two?

Interestingly, `decipher(data, output)` is the only function that uses the `length` field in the header. All other functions are coded in such a way that knowing this length is not required. This causes another peculiar observation: when setting `length` to zero, no decryption takes place, and the packet is processed as-is. This observation is not required to solve the challenge though (but it does make it easier to construct packets).

Authentication and Commands

The above two vulnerabilities are sufficient to exploit the challenge. However we had some problems doing this, and instead opted for another approach. Our approach relies on additional functionality of the challenge: letting it execute commands. We can execute three commands: (1) getting the current server time; (2) reading an arbitrary file; (3) executing a system command. The latter two commands required the packet to be authenticated.

The client must provide a checksum equal to `H(H(password||key) || H(data||password))` for the packet to be authenticated. Here `password` is a secret loaded at startup, and `key` and `data` are chosen by the client. The function `H()` seems to be a custom (and insecure?) hash function.

Exploitation

We use both length-check vulnerabilities to read the secret `password`. Once we have this, we can execute arbitrary system commands on the server, and exploitation becomes trivial.

In the `decipher(data, output)` function we overwrite the return address. This is done by first overwriting the local variable `loop`, such that the next xor-decrypt operation will point to the return address. We prevent other variables from being overwritten by XORing with zero. Recall that we can XOR 8 bytes in total, and in particular we XOR these bytes with the `key` in the header of the packet.

Where will we point the return address to? First observe that, because of the last `mempcy` call, the `eax` register contains a pointer to the local buffer when returning from the function. Hence the content pointed to by `eax` is under our control. If we now exploit the binary for interesting gadgets, we spot the following:

```
.text:08049290                mov     eax, [ebp+packet]
.text:08049293                add     eax, 8
.text:08049296                sub     esp, 8
.text:08049299                push    eax             ; filename
.text:0804929A                push    [ebp+fd]        ; fd
.text:0804929D                call    read_file
```

This is the code that calls `read_file(sockfd, name)`. When returning from `decipher` the `ebp` register is restored, hence it correctly points to the local variables. However, variable `packet` is not yet initialized by the program, hence we can't use it. Instead we redirect code to `push eax; push [ebp+fd]`.
Since `eax` points to the beginning of the buffer this code will successfully be executed. Putting the string`password.txt` in the start of the buffer will now make the service print out the password **pass123**.
We can now send authenticated commands and execute arbitrary shell commands. The flag was in the file`flag.txt` in the same directory as `password.txt` (i.e. the current working directory).

Remark: the problem we encountered when directly calling `system(cmd)` was that it would overwrite the local buffer used in `decipher` (since it was saved on the stack). Recall that the pointer to this buffer was stored in `eax`. Luckily, in the `read_file` function, the stack is used in such a way so this didn't occur.