CS-C3130 Information Security (2021)
Exercise 4

IoT Insecurity: Buffer Overrun

In this exercise, you will target an Internet-of-Things (IoT) device that can receive simple authenticated commands at a TCP port. You may need to familiarize yourself with the C language and byte manipulation. Part A is easy, part B can be more difficult, and part C is intended as a challenge for true hackers. As before, the exercise launcher is running at `https://infosec1.vikaa.fi/`.

Part A. Crash the device (10p)

While various mitigation techniques have reduced the number of serious buffer overruns in desktop and server software, low-end devices in the Internet of Things may still be vulnerable because of the need to minimize the code footprint.

The target in this exercise is an IoT device that is connected to the public internet so that it can be controlled remotely. See the exercise launcher for the device address and port number. It listens for connections on a TCP port and accepts command messages. If you try to send commands to the device, they will be rejected. You can nevertheless send inputs and see how the device reacts. If you can make the device crash or reboot, it is often symptom of further security flaws such as buffer overrun.

Your goal in part A is to exploit a buffer overrun and cause the device to crash or reboot. To do this, experiment with the service, learn its features (see part B for hints if you get stuck), and look for data fields that accept longer inputs than they should. When you find one, send gradually longer inputs until the system hopefully crashes.

When you do not have physical access to the device, it is not always easy to detect when the device crashes or reboots. It might just stop responding or close the connection. Fortunately, the target device for this exercise is in our laboratory and the exercise launcher can confirm the crash for you — at least in most cases.

You can open a TCP connection to the server and send with the `netcat` tool (get the port number from the exercise launcher):

```
nc device1.vikaa.fi 34567 < file.txt
echo "abcdefghijklmn" | nc device1.vikaa.fi 34567
```

Initially, you can also experiment with `telnet`, which opens an interactive TCP connection to the given port in the server and allows you to send text from the keyboard. After a while, it is easier to use `nc`.

Unfortunately, the `telnet` that you can enable as a Windows feature does not work well for this exercise. You can access `nc` and `telnet` from the bash shell in Ubuntu for Windows (install from Microsoft store) or from a Linux VM. Another workaround is to use the built-in SSH client in Windows to connect to a university server, e.g. `ssh username@brute.aalto.fi` (use your Aalto username and password) and to launch `nc` and `telnet` from there. For the Windows command line, use `PowerShell`, not the legacy `cmd.exe`. That typically avoid some additional problems.

Part B. Exploit buffer overrun (10p)

In part B, your goal is to get the device to accept a command. We do not really mind which command you send to the device as long as the device accepts it as authentic. For real attackers, this is naturally only the first step. Once they learn to forge commands, they can do much more.

The device accepts commands that are authenticated with a message authentication code (HMAC-SHA256). However, you do not know the secret key for computing the HMAC and must, therefore, exploit the buffer overrun vulnerability found in part A. Crafting an exploit that achieves something useful is *much harder* than just crashing the target.

Here are some more tehnical details that will be useful. The format of the command messages is the following:

`<user>;<command>;<hex_hmac><lf>`

The message ends in a linefeed or carriage return character. You could write a client program that sends the crafted message via a stream socket. In this case, an easier option is to use `nc` to send a file or construct the message on the command line. For example:

```
echo "151055;aaa;b0bf94d...373c999c76d9" | nc device1.vikaa.fi 34567
```

The HMAC is in hexadecimal format and computed from the following concatenation:

`ClientCmd|<user_number>|<command>`

For example:

```
ClientCmd|151055|aaa
```

The HMAC input includes the bar symbols. The constant string "ClientCmd" in the beginning is a type tag indicates the type of the message. In order to compute the HMAC, you could use the OpenSSL library and the function HMAC . Make sure not to include any additional whitespace such as a trailing newline in the HMAC input. Note that you should send the hexadecimal representation of the computed HMAC to the server.

Here is the relevant part of the server C code. Error handling and other irrelevant parts have been omitted.

```c
#define KEYPHRASE "This is a dummy key!" /* TODO: do not store keys in the source code */
#define KEYLEN 20
#define DELIMETERS ";"
#define MAXCMDLEN 18
#define MAXUSERLEN 10
#define MAXREPLYLEN 80
#define HMACLEN 32 /* 32 bytes = 64 hex characters */
#define CLIENTCMDTAG "ClientCmd"
#define MAXHMACINPUTLEN (sizeof(CLIENTCMDTAG) + 1 + MAXCMDLEN + 1 + MAXUSERLEN) /* 40 */

void handle_message(int connectiondf, char *msg)
{
    char *command_rec;
    char *user_rec;
    char *hmac_rec;
    unsigned char *hmac;
    char keyphrase[KEYLEN + 1] = {0};
    char hmac_input[MAXHMACINPUTLEN + 1] = {0};
    char hex_hmac[HMACLEN * 2 + 1] = {0};
    char reply[MAXREPLYLEN + 1] = {0};
    int keylen = KEYLEN;
    int i;

    user_rec = strtok(msg, DELIMETERS);
    command_rec = strtok(NULL, DELIMETERS);
    hmac_rec = strtok(NULL, "\n");

    /* ... SOME MSG SYNTAX CHECKS OMITTED ... */

    /* prefix tag to HMAC input */
    sprintf(hmac_input, "%s|%s|%s", CLIENTCMDTAG, user_rec, command_rec);

    /* compute HMAC */
    hmac = HMAC(EVP_sha256(), keyphrase, keylen, (const unsigned char *)hmac_input,
                (int)strlen(hmac_input), NULL, NULL);
    if (!hmac)
    {
        fprintf(stderr, "%s", "Server: HMAC failed\n");
        exit(EXIT_FAILURE);
    }

    /* hexadecimal representation of the HMAC computed above */
    for (i = 0; i < HMACLEN; i++)
        sprintf(hex_hmac + i * 2, "%02x", hmac[i]);

    /* check if HMAC is correct */
    if (!memcmp(hex_hmac, hmac_rec, HMACLEN * 2))
    {
        snprintf(reply, MAXREPLYLEN, "Authentication successful. Processing command.\n");
        reply_to_client(connectiondf, reply);

        /* ... COMMAND PROCESSING HAPPENS HERE ... */

    }
    else
    {
        snprintf(reply, MAXREPLYLEN, "Invalid HMAC. Command not accepted.\n",
                 command_rec);
        reply_to_client(connectiondf, reply);
```

```
    }
}
```

In the code, the `hmac_input` array is defined right after the `keyphrase` variable. If we look at the memory dump of the stack on the IoT device, we notice that the local variables are laid out in the call-stack memory in the reverse order. Thus, by supplying a long enough `command`, you could overwrite the `keyphrase`. This just might enable an attacker to send commands to the device.

```
0x7ffd49144520:  10 43 a3 01 00 00 00 00  20 00 00 00 14 00 00 00   .C......  ........
0x7ffd49144530:  49 6e 76 61 6c 69 64 20  48 4d 41 43 2e 20 43 6f   Invalid  HMAC. Co
0x7ffd49144540:  6d 6d 61 6e 64 20 22 64  75 4d 70 22 20 6e 6f 74   mmand "d uMp" not
0x7ffd49144550:  20 61 63 63 65 70 74 65  64 2e 0a 00 00 00 00 00    accepte d.......
0x7ffd49144560:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0x7ffd49144570:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0x7ffd49144580:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0x7ffd49144590:  39 32 63 64 34 38 66 61  35 38 66 63 66 63 65 39   92cd48fa 58fcfce9
0x7ffd491445a0:  33 38 31 36 63 34 39 31  31 38 61 65 66 63 62 66   3816c491 18aefcbf
0x7ffd491445b0:  62 63 36 39 39 33 65 61  33 30 38 61 61 36 64 62   bc6993ea 308aa6db
0x7ffd491445c0:  34 64 61 63 35 33 35 37  31 35 66 37 39 31 32 38   4dac5357 15f79128
0x7ffd491445d0:  00 00 00 00 00 00 00 00  0f cc d5 fb a8 7f 00 00   ........ ........
0x7ffd491445e0:  43 6c 69 65 6e 74 43 6d  64 7c 31 35 31 30 35 35   ClientCm d|151055
0x7ffd491445f0:  7c 61 61 61 00 00 00 00  00 00 00 00 00 00 00 00   |aaa.... ........
0x7ffd49144600:  00 00 00 00 00 00 00 00  00 d3 d5 fb a8 7f 00 00   ........ ........
0x7ffd49144610:  54 68 69 73 20 69 73 20  61 20 64 75 6d 6d 79 20   This is  a dummy
0x7ffd49144620:  6b 65 79 21 00 7f 00 00  16 33 40 00 00 00 00 00   key!.... .3@.....
0x7ffd49144630:  e0 f3 4e fc a8 7f 00 00  9c 46 14 49 fd 7f 00 00   ..N..... .F.I....
0x7ffd49144640:  97 46 14 49 fd 7f 00 00  90 46 14 49 fd 7f 00 00   .F.I.... .F.I....
0x7ffd49144650:  88 2f 40 00 00 00 00 00  50 2f 40 00 00 00 00 00   ./@..... P/@.....
```

Note 1: The call stack grows *downwards* in the process addresses space, which is *up* in the memory listing above. The ordering of the local variables depends on the C compiler. Note 2: Due to alignment of variables to word boundaries, there can be empty space between variables. Note 3: The memory alignment and endianness of integers may be different if you compile the code on your own system. Note 4: Strings in C are null terminated. That is, a string ends in the first zero byte in the memory, and if you overwrite the byte with something other than zero, string-processing functions will think the string continues until the next zero in the memory. Note 5: The `sprintf()` function places a zero byte immediately after its output string in the memory to indicate that the output string ends there.

Armed with this knowledge, your goal is to craft a message to the device in such a way that the device accepts its authentication.

The buffer overrun vulnerability in this exercise is caused by a simple software bug (missing bounds check). However, the device also has a serious protocol-level design flaw. If you have listened to the the lecture on security protocols, you may notice that there is no replay protection for the authenticated message above. Thus, captured old messages can be sent again to the device. Many weak links in the security of this device!

Part C (bonus problem). Crack the key (10p)

In part C, your goal is to discover the keyphrase. You probably completed part B without learning the server's secrets. It turns out that there is a way to discover the keyphrase as well.

It is impossible to brute-force guess a 20-byte key as a whole. However, security flaws sometimes enable you to brute-force guess a small part (e.g. one byte) of the key. You need to be able to check the correctness or incorrectness of the guess, for example, by observing or not observing a crash. If that is the case, you may be able to discover the whole secret byte by byte. To get started, you should first craft such an input that the resulting device output or behavior gives you information about one byte of the keyphrase.

For this attack, you will need to write code to send the malicious inputs to the target. Do not overload the target with a brute-force attack; it is not necessary and will not help in solving the exercise.