Advanced Persistent Jest

... I wield the power of business excellence, ten thousand senior project managers could not manage me.

Writeup – smartfridge1 and smartfridge2 (33C3 Part 2 of 2)

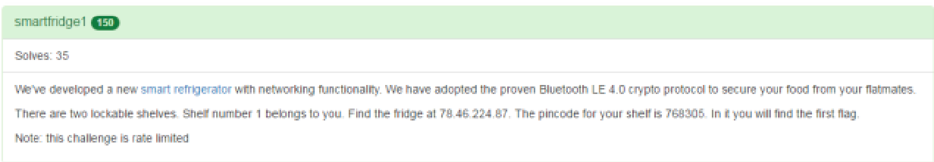
Posted on December 30, 2016 by Norman

During the last 2 days, I participated in the 33c3 CTF. This post contains the second part of my writeups for this CTF, covering the smartfridge1 and smartfridge2 challenges.

Note that the content provided for smartfridge2 (a crypto challenge!) speeds up solving smartfridge1 considerably, so I will provide the original files for both challenges together.

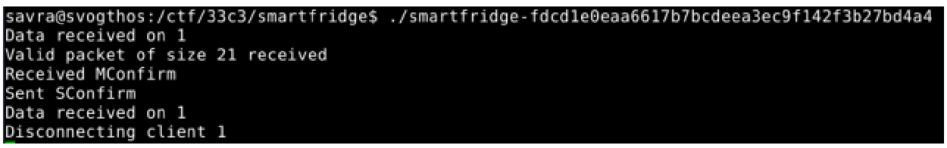
smartfridge1

These two challenges were presented as a binary (smartfridge1), and a pcap of someone communicating with said binary (smartfridge2), which you can download here. The challenge text indicates that this is something to do with the Bluetooth LE 4.0 protocol:

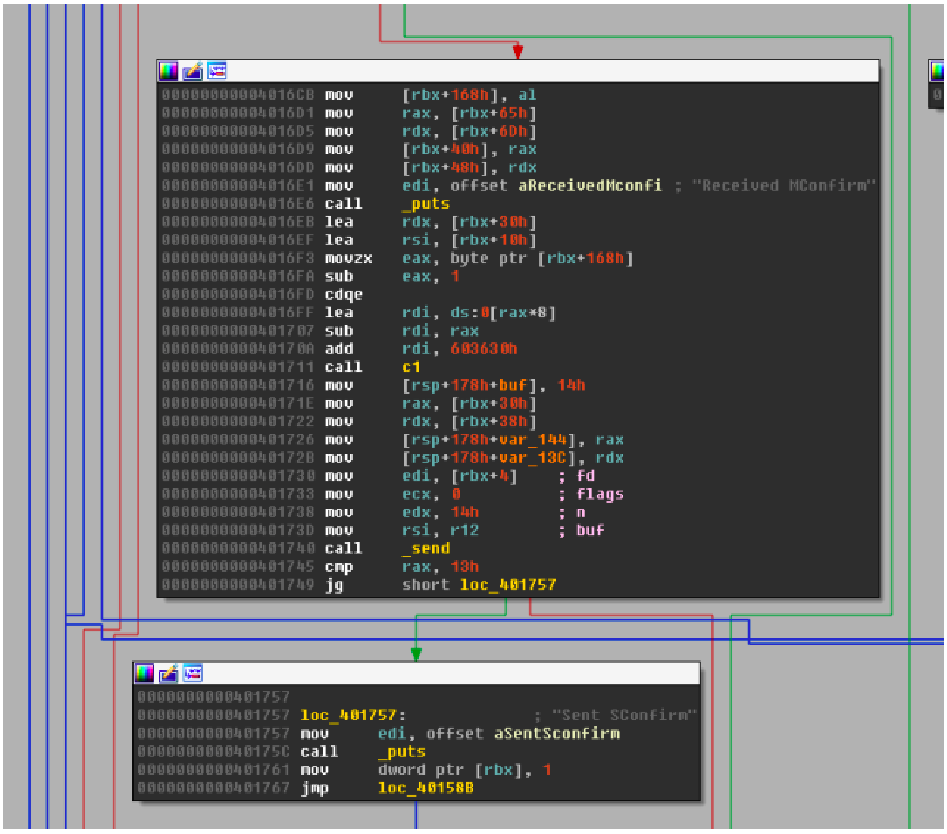


Our task appears to be to reverse engineer the provided binary, build something which can interface with it (and thus, the remote server), and send a series of commands to “retrieve” the flag.

On first inspection, we notice that the binary makes heavy use of OpenSSL AES functions, and is littered with printf debugging. As a starting point, I decided to use the pcap provided for smartfridge2, and simply send the first packet in the pcap to the binary. Instant success:



We can look for these printf statements in IDA (the second SEND below is an SCONFIRM packet):



Our next step is to carefully gdb our way through the "c1" function, which appears to set an AES encryption key and encrypt *something*, to determine what's being encrypted:

```
=> 0x401481 <e+56>:     call   0x400cb0 <AES_encrypt@plt>
   0x401486 <e+61>:     mov    rax,QWORD PTR [rsp+0xf8]
   0x40148e <e+69>:     xor    rax,QWORD PTR fs:0x28
   0x401497 <e+78>:     je     0x40149e <e+85>
   0x401499 <e+80>:     call   0x400d40 <__stack_chk_fail@plt>
Guessed arguments:
arg[0]: 0x608020 --> 0xe60b7cb4bf1804a
arg[1]: 0x608040 --> 0x0
arg[2]: 0x7fffffffdec0 --> 0x1e240
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdec0 --> 0x1e240
0008| 0x7fffffffdec8 --> 0x0
0016| 0x7fffffffded0 --> 0x6362812263628122
0024| 0x7fffffffded8 --> 0x6362812263628122
0032| 0x7fffffffdee0 --> 0x93fbaa0ef0992b2c
0040| 0x7fffffffdee8 --> 0x93fbaa0ef0992b2c
0048| 0x7fffffffdef0 --> 0xc8be8e8a5b452484
0056| 0x7fffffffdef8 --> 0xabdc0fa83827a5a6
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000401481 in e ()
gdb$
```

In this instance, the key is somewhat recognizable (the hexadecimal representation of atoi("123456" – which appears to be a test PIN)), but the plaintext (0xe6...) is not. It doesn't help that despite sending a static message to the server, this plaintext seems to change every time. Let's set this aside for now.

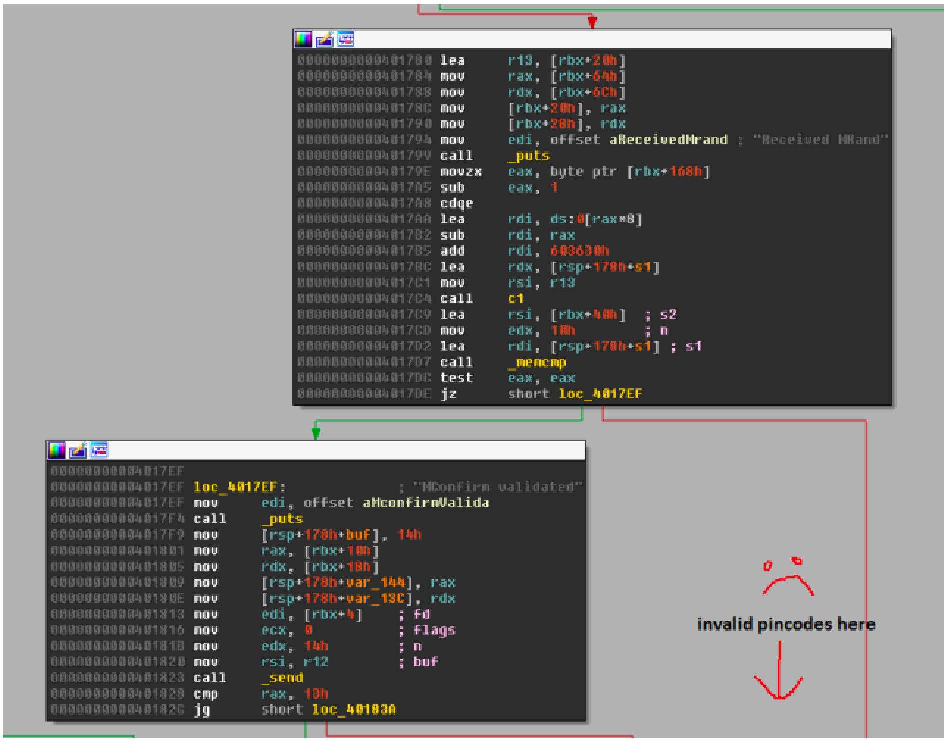
Going back through the disassembly, we can shed some light on the structure of the packet: breakpoints at 0x40158B and 0x40159A reveal that the first DWORD is a packet length, which must be above 3 and below 84. Also, we can see a "state variable" being used to control program flow – packets must be sent in order:

```
00000000004015B4 mov     edx, r13d
00000000004015B7 mov     esi, offset aValidPacketOfS ; "Valid packet of size %d received\n"
00000000004015BC mov     edi, 1
00000000004015C1 mov     eax, 0
00000000004015C6 call    ___printf_chk
00000000004015CB cmp     dword ptr [rbx], 1 ; is device paired?
00000000004015CE jbe     loc_40166B
```

Continuing with this avenue of investigation, we add in the second packet from the pcap (the one starting with “\x14\x00\x00\x00”) to our client:

```
savra@svogthos:/ctf/33c3/smartfridge$ ./smartfridge-fdcd1e0eaa6617b7bcdeea3ec9f142f3b27bd4a4
Data received on 1
Valid packet of size 21 received
Received MConfirm
Sent SConfirm
Data received on 1
Valid packet of size 20 received
Received MRand
Invalid pincode
Disconnecting client 1
```

This time, something more interesting comes up: once more, we return to the disassembly to find out what’s causing our “invalid pincode” message:



Here, we go back to dynamic analysis to find what’s being passed to c1, and what it’s being compared against. To make life easier for ourselves, we replace the “body” of each packet with clearly recognizable strings like “\x01″*16 (MConfirm, packet 1) and “\x02″*16 (MRand, packet 2).

This time, we have some success:

```
=> 0x4017c4 <handle+624>:        call   0x4014a8 <c1>
   0x4017c9 <handle+629>:        lea    rsi,[rbx+0x40]
   0x4017cd <handle+633>:        mov    edx,0x10
   0x4017d2 <handle+638>:        lea    rdi,[rsp+0x20]
   0x4017d7 <handle+643>:        call   0x400d70 <memcmp@plt>
Guessed arguments:
arg[0]: 0x603637 --> 0x363534333231 ('123456')
arg[1]: 0x608030 --> 0x202020202020202
arg[2]: 0x7fffffffe040 --> 0x7fffffffe070 --> 0x7ffff763f130 --> 0xc0022000015b5
```

Here, we can see that upon receiving the “MRand” packet, “c1” is called to encrypt “\x02\x02\x02\x02…” with “123456”, our test PIN code, the result of which is “d86d0cfbc7bb0343f09fad5ca6824429”. We then proceed to the memcmp call at 0x4017D7:

```
=> 0x4017d7 <handle+643>:       call   0x400d70 <memcmp@plt>
   0x4017dc <handle+648>:       test   eax,eax
   0x4017de <handle+650>:       je     0x4017ef <handle+667>
   0x4017e0 <handle+652>:       mov    edi,0x402037
   0x4017e5 <handle+657>:       call   0x400c20 <puts@plt>
Guessed arguments:
arg[0]: 0x7fffffffe040 --> 0x4303bbc7fb0c6dd8
arg[1]: 0x608050 --> 0x101010101010101
arg[2]: 0x10
```

Hang on, we know those values! It's comparing the content of our MConfirm packet with AES(key="123456",content=MRand)! Knowing this, it is trivial to pass the MRand check, by sending the correct MConfirm packet first:

```
savra@svogthos:/ctf/33c3/smartfridge$ ./smartfridge-fdcd1e0eaa6617b7bcdeea3ec9f142f3b27bd4a4
Data received on 1
Valid packet of size 21 received
Received MConfirm
Sent SConfirm
Data received on 1
Valid packet of size 20 received
Received MRand
MConfirm validated
Sent SRand
Paired
Data received on 1
Disconnecting client 1
```

Success! At this point, we are sending:

- Packet 1 – MConfirm: \x15\x00\x00\x00\x00\x02 + aes(key="123456",text="\x02"*16)
- Packet 2 – MRand: \x14\x00\x00\x00\x00 + "\x02" * 16

Now that we're paired, let's try sending another packet. As the process logic appears to be controlled via state variable instead of any "command specifier", let's just try sending a packet containing "\x03"*16 and let's see what happens:

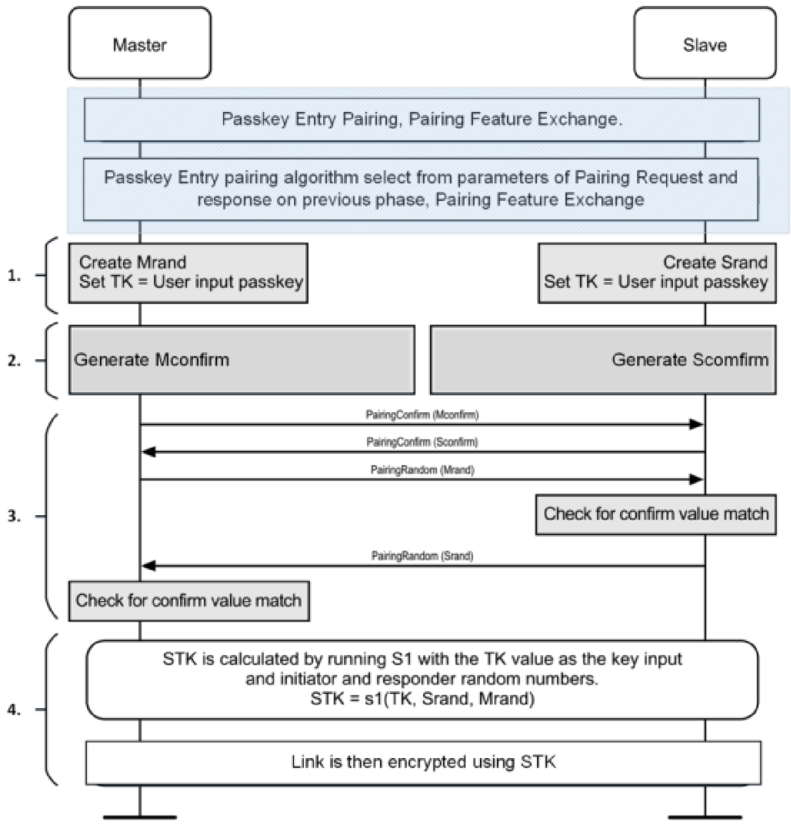
```
Paired
Data received on 1
Valid packet of size 20 received
Invalid padding
Disconnecting client 1
```

Going back through the disassembly, we can see that this comes from a decryption function at 0x4015F2:

```
00000000004015F2
00000000004015F2 loc_4015F2:
00000000004015F2 lea     r15, [rbx+0E8h]
00000000004015F9 lea     rdi, [rbx+50h]
00000000004015FD mov     rcx, r15
0000000000401600 mov     edx, ebp
0000000000401602 mov     rsi, r14
0000000000401605 call    decrypt
000000000040160A lea     eax, [r13-5]
000000000040160E movzx   eax, byte ptr [rbx+rax+0E8h]
0000000000401616 cmp     al, 10h
0000000000401618 jbe     short loc_40162E
```

At a glance, this function appears to attempt an AES decryption, and if the decryption is successful, the decrypted data gets passed to strtok (0x40162E), and from there, to command handlers which try to do string comparisons against known commands (e.g. "OPEN" at 0x4018B3), giving us some clue as to what the plaintext needs to be.

Stepping through this decryption function, I quickly noticed that the key was not the "PIN code" key. Perplexed, I turned to the Bluetooth Low Energy documentation:



from https://blog.bluetooth.com/bluetooth-pairing-part-3-low-energy-legacy-pairing-passkey-entry

From here, we can make a reasonable guess that the decrypt() function is using the STK instead of the TK / pincode. After reading through the documentation, we can determine that the STK is calculated thusly:

```
STK = AES(key=pin,text=second_half_of_srand + first_half_of_mrand)
```

Going through the disassembly, we can see that the “s1” function is present in the binary, at 0x40188F – we'll need to emulate this function in our client, so we can use the same STK session key to encrypt command traffic to the server. In gdb:

```
=> 0x40188f <handle+827>:        call   0x40150a <s1>
   0x401894 <handle+832>:        mov    DWORD PTR [rbx],0x2
   0x40189a <handle+838>:        mov    edi,0x402065
   0x40189f <handle+843>:        call   0x400c20 <puts@plt>
   0x4018a4 <handle+848>:        jmp    0x40158b <handle+55>
Guessed arguments:
arg[0]: 0x7fffffffe030 --> 0x1e240
arg[1]: 0x608020 --> 0xd1305cafc8590ca9
arg[2]: 0x608030 --> 0x202020202020202
arg[3]: 0x608060 --> 0x0
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe020 --> 0x7fffffffe050 --> 0xc8590ca900000014
0008| 0x7fffffffe028 --> 0x7fffffffe040 --> 0x4303bbc7fb0c6dd8
0016| 0x7fffffffe030 --> 0x1e240
0024| 0x7fffffffe038 --> 0x0
0032| 0x7fffffffe040 --> 0x4303bbc7fb0c6dd8
0040| 0x7fffffffe048 --> 0x294482a65cad9ff0
0048| 0x7fffffffe050 --> 0xc8590ca900000014
0056| 0x7fffffffe058 --> 0xf6d83fbfd1305caf
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x000000000040188f in handle ()
gdb$ x/32bx 0x608020
0x608020:       0xa9    0x0c    0x59    0xc8    0xaf    0x5c    0x30    0xd1
0x608028:       0xbf    0x3f    0xd8    0xf6    0x62    0x67    0xea    0x7f
0x608030:       0x02    0x02    0x02    0x02    0x02    0x02    0x02    0x02
0x608038:       0x02    0x02    0x02    0x02    0x02    0x02    0x02    0x02
gdb$
```

Stepping through this function, we can confirm the documentation, and thus, correctly emulate s1 and derive our own STK (see the solution Python script).

With the STK, we can now send correctly encrypted commands to the server. The commands "OPEN 2", "LIST" and "TAKE 0" can be used to retrieve a flag – in our test binary, this is "33C3_NOT_FLAG_2" – the only remaining change is to modify the PIN code, and send traffic to the actual server instead:

```
[+] Opening connection to 78.46.224.87 on port 12345: Done
 [>] sending MCONFIRM, which is AES(key=PIN,data=MRAND) : 823d6a280880d28de56e16d4c1ab3fab
 [<] recving SCONFIRM, decrypted key in data_decrypted...
 [!] received: caf440ad6d1b6ad064f20db5d10076bc
 [!] decrypted: 0090492d2f9ef5f78980176b864cd9b4
 [>] sending MRAND, which is 0x02 * 16
 [<] recving SRAND, decrypted key in srand_decrypted...
Traceback (most recent call last):
  File "./client.py", line 47, in <module>
    data = p.recv()
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 75, in recv
    return self._recv(numb, timeout) or ''
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 154, in _recv
    if not self.buffer and not self._fillbuffer(timeout):
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 125, in _fillbuffer
    data = self.recv_raw(4096)
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/sock.py", line 52, in recv_raw
    raise EOFError
EOFError
[*] Closed connection to 78.46.224.87 port 12345
```

A little trial and error later, and I realized that the failed pairing was due of the initial MConfirm message. When we prefixed the packet with "\x15\x00\x00\x00\x02", we were selecting user 2, who seemed to have a different PIN code. Moments later (and after changing "OPEN 2" to "OPEN 1", the flag:

```
 [>] sending MCONFIRM, which is AES(key=PIN,data=MRAND) : 823d6a280880d28de56e16
d4c1ab3fab
 [<] recving SCONFIRM, decrypted key in data_decrypted...
 [!] received: 64d88ee82813ade6aed781d42b41eead
 [!] decrypted: 912177ad4191e02beda9befc2049fe53
 [>] sending MRAND, which is 0x02 * 16
 [<] recving SRAND, decrypted key in srand_decrypted...
 [!] SRAND received: 912177ad4191e02beda9befc2049fe53
 [!] our unencrypted session key is [CORRECT] : eda9befc2049fe530202020202020202
 [!] our session key is s1(tk,srand,mrand) : ae7f3d22690b1e1b9b3acd375c7298a1
 [CBC_KEY/session key IS CORRECT:BREAK AT 0x401894 and VERIFY 0x608060]
 [>] trying to send OPEN 1 1, which is afb443ae8bcb03b04747e133df644440
 [>] trying to send LIST, which is 50fab3818df1257ad1cdc1e5b71847d7
0. toast
1.
2.
3.
4.
\x07\x07\x07\x07\x07\x07\x07
 [>] trying to send TAKE 0, which is 42f67ffdfad38215b04d685815e9ded6
toast: 33C3_s1impl3_4sync_s3rver
\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f
[*] Closed connection to 78.46.224.87 port 12345
savra@svogthos:/ctf/33c3/smartfridge$
```

You can find the complete solution script here – it's a functional implementation of Bluetooth LE pairing over TCP!

smartfridge2

This challenge was presented as a PCAP (download together with smartfridge1 binary above), and we were tasked with finding the flag belonging to "user 2" on another shelf in the smart fridge.

With the knowledge of the Bluetooth LE pairing protocol gained during the last challenge, this challenge was trivial. We had a PCAP of a Bluetooth pairing, for "user 2" (as indicated by "\x15\x00\x00\x00\x02" in the MConfirm packet).

With the MConfirm and MRand packets, and knowing that:

```
MConfirm = AES(key=pin,cleartext=MRand)
```

it is trivial to brute force the key. A few modifications to the client script later, and we have the second flag:

```
 [!] received: 0cbc663729f9fe6991f1400b07014ae8
 [!] decrypted: 5ac34b0f6ebbe57c02d5a43d5aeef0e0
 [>] sending MRAND, which is 0x02 * 16
 [<] recving SRAND, decrypted key in srand_decrypted...
 [!] SRAND received: 5ac34b0f6ebbe57c02d5a43d5aeef0e0
 [!] our unencrypted session key is [CORRECT] : 02d5a43d5aeef0e00202020202020202
 [!] our session key is s1(tk,srand,mrand) : d6c718ed4931b9524c7521e284e4b834
 [CBC_KEY/session key IS CORRECT:BREAK AT 0x401894 and VERIFY 0x608060]
 [>] trying to send OPEN 2 1, which is 5b22da7147d08d7a57449a4574c63c66
 [>] trying to send LIST, which is 86e2456f53d674c9b499c82270082545
0. yoghurt
1.
2.
3.
4.
\x05\x05\x05\x05\x05
 [>] trying to send TAKE 0, which is 7a64ba5f9473c1a9ecc02b08b8d2cd8f
yoghurt: 33C3_sh0rt_p4ssc0de_1s_sh0rt
```

You can find the modified solution script here.

As always, I'd like to thank the organisers of 33c3's CTF, Eat Sleep Pwn Repeat, for putting together a fantastic event. While I was not able to solve as many challenges as I had hoped, I had a lot of fun and learned alot solving the ones I was able to complete, and look forward to doing the others in my own time.

If I might offer some feedback, the point values seem imbalanced, and this was reflected by other participants in this event. For example, it is clear that smartfridge2 required a miniscule fraction of the effort required by smartfridge1, yet was worth 50% of the points. Still, this by no means detracted from the excellence of the event, as it didn't make the challenges less fun.

A safe and happy New Year's to you all, and see you all in a few weeks for the Insomni'hack Teaser CTF.

About Norman
Sometimes, I write code. Occasionally, it even works.
View all posts by Norman →

This entry was posted in Bards, Computers, Jesting and tagged bluetooth, business excellence, glorious victory. Bookmark the permalink.

Advanced Persistent Jest