Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Audit-Report CrypTech/DiamondKey 09.2018

Cure53 & Trustworks, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Kobeissi, MSc. N. Krein, MSc. D. Weißer, BSc. J. Hector, Dr. M. Müllner, Dr. M. Kammerstetter

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"The goal of the CrypTech project is to create an open-source hardware cryptographic engine that can be built by anyone from public hardware specifications and open-source firmware and operated without fees of any kind. The team working on the project is a loose international collective of engineers trying to improve assurance and privacy on the Internet. It is funded diversely and is administratively quartered outside the US.

The project solicits functional requirements from a wide range of organizations. It will focus on the classic low level cryptographic functions and primitives, and not get drawn into re-implementation of application protocol layers."

From https://cryptech.is

This report documents the results of a large-scale security assessment of the CrypTech/ DiamondKey HSM firmware. The project was carried out in September 2018 by a joint team comprising members of Cure53, Berlin, and Trustworks, Vienna. It entailed a source code audit, a cryptography review and additional, relevant tests against the aforementioned compound. Nine security-relevant discoveries were made by the testing team.

In terms of resources and timeline, the project team included six testers from Cure53 and two engineers representing Trustworks. Once the agreements on the scope and schedule were reached, the testing team was granted a time budget of thirtytwo days, which had then been invested into investigations completed through a range of varied approaches. In addition, some of the budgeted days were allocated to communications and coordination of the actual assessment. The vast majority of work took place in September 2018, specifically in Calendar Weeks (CWs) 38 and 39.

Moving on to the approach, a white-box methodology was used for this assessment. This means that the testers had been provided with access to all relevant sources, as well as benefitted from several test-devices sent by the customer for the testing purposes. In addition, the CrypTech/DiamondKey teams organized several walkthroughs in order to makes sure that the intricate and complex nature of the scope is fully understood. In relation to the complexity, the assessment was divided into six Work Packages (WPs), each with clear objectives and targets. While the WP1 encompassed Documentation Review and WP2 entailed investigations of the System Build/Boot, the remaining four WPs pertained to audits of the demarcated areas of interest. Specifically, these were the Verilog Audit (WP3), C/Verilog Interface Audit (WP4), General C Audit (WP5), and, last but not least, the Cryptography Audit (WP6).

Given the excellent product overviews and walkthroughs that the Cure53 and Trustworks teams received prior to the start of the assessment, the need for clarifications or contact during the actual project was not overly pronounced. For the most part, Cure53 provided status updates to the CrypTech/DiamondKey teams and only posed very few minor questions. Live test-related communications were done on several, dedicated and

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

shared Slack channels. One channel was used to coordinate the work between Cure53 and Trustworks, while the other served as a direct line between the testers and the in-house teams at CrypTech/DiamondKey. With this setup, the project progressed well and without any major delays, even in the face of several test devices arriving late at their final destinations due to logistical problems.

The assessment was successful in so far that it revealed a number of issues that the CrypTech/DiamondKey teams clearly needs to tackle. Among the aforementioned nine discoveries, five were categorized as actual security vulnerabilities and four belong to the class of general, less-severe weaknesses. It is vital to underline that all vulnerabilities had greatly elevated rankings, meaning that the risks they posed were significant. Three problems were ascribed with the highest-possible "*Critical" severity,* while the remaining two vulnerabilities were deemed to signify "*High"*-level, substantial threats. Foreshadowing some of the conclusions, this clearly indicates that - from a security perspective - the results are not satisfactory when seen vis-à-vis the CrypTech/ DiamondKey HSM firmware's goals.

In the following sections, this report will first shed light on the items in scope of this assessment. Then, the findings will be described in a considerable, technical detail on a case-by-case basis. Whenever possible, the testing team also furnishes advice on possible mitigation routes for the spotted and distinct problems. Lastly, the report shares some general impressions gained by the Cure53 and Trustworks testing team, commenting on the overall security posture of the CrypTech/DiamondKey HSM compound.

Scope

- **CrypTech/DiamondKey Firmware**
  - Cure53 was tasked with auditing the code for the CrypTech/DiamondKey firmware; full access to all relevant sources was granted.
  - Cure53 was also supplied with several test-devices to directly work on.
  - The teams also go access to the extended documentation pertaining to the inner-workings of the firmware. This facilitated reaching a better coverage during this audit.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *CT-01-001*) for the purpose of facilitating any future follow-up correspondence.

CT-01-005 MCU: OOB writes through dynamic stack allocations *(**Critical**)*

Throughout the code, multiple locations use untrusted user-input to dynamically allocate memory on the stack. This can lead to out-of-bound (OOB) writes and therefore memory corruptions, effectively due to overly large parameters. Below is an example code excerpt that depicts vulnerability to this issue. An attacker can trigger an OOB write by providing a negative value for *attributes_len*.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_server.c

An Example of Affected Code:

```
static hal_error_t pkey_get_attributes(const uint8_t **iptr, const uint8_t *
const ilimit, uint8_t **optr, const uint8_t * const olimit)
{
    hal_client_handle_t client;
    hal_pkey_handle_t pkey;
    uint32_t attributes_len, u32;
    uint8_t *optr_orig = *optr;
    hal_error_t err;

    check(hal_xdr_decode_int(iptr, ilimit, &client.handle));
    check(hal_xdr_decode_int(iptr, ilimit, &pkey.handle));
    check(hal_xdr_decode_int(iptr, ilimit, &attributes_len));

    hal_pkey_attribute_t attributes[attributes_len > 0 ? attributes_len : 1];

    for (size_t i = 0; i < attributes_len; i++)
        check(hal_xdr_decode_int(iptr, ilimit, &attributes[i].type));
[...]
```

In a low-level assembly, a dynamic stack allocation as highlighted above, is simply subtracting a value from the stack pointer. Since the size parameter, controlled by the attacker, is a 32-bit value, it is possible to move the stack pointer to anywhere in the address space. Using the decode operation in the last for-loop, an attacker can achieve an arbitrary memory write and, ultimately, gain arbitrary code execution.

It is recommended to sanitize and validate all user-input, especially when used in the context of critical operations such as memory allocation. The above code excerpt is just one example that provides ideal conditions for exploitation. It is further recommended to review other parts of the code that follow a similar pattern and apply the same fix.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Note: *This issue was identified during a manual code audit but not yet verified on a running system. Given the time constraints of this assessment, the team opted for an increased coverage instead of clean reproducibility.*

CT-01-006 MCU: Value cast allows a bypass of the size checks (*Critical*)

The decode functions, which are used to unserialize data from the input request, preform size checks to ensure that enough data is indeed available to unserialize the given type. For some types, the *length* parameter is user-controlled, for example when unserializing a string. These size checks can be bypassed due to the cast to a signed int.

Affected File:
Source/Cryptech/files/Source/Cryptech/releng/alpha/source/sw/libhal/xdr.c

Affected Code:
```
hal_error_t hal_xdr_decode_fixed_opaque_ptr(const uint8_t ** const inbuf, const
uint8_t * const limit, const uint8_t ** const value, const size_t len)
{
[...]
    /* buffer overflow check */
    if (limit - *inbuf < (ptrdiff_t)len)
        return HAL_ERROR_XDR_BUFFER_OVERFLOW;
[...]
```

Highlighted above is the check in question, which normally ensures that enough data is available to unserialize *len* amount of bytes. Using a negative value for *len* will bypass the check and potentially lead to buffer overflows or out-of-bound (OOB) writes.

To verify that the compare operation is signed, a small application was written to replicate the behavior and inspected it with a disassembler (replicated on an *ARM* architecture). The list below shows the relevant C++ source and the connected disassembly.

PoC Application (C++):
```
#include <cstdio>
#include <cstdlib>
#include <string.h>
#include <stdint.h>

int main(int argc, char *argv[])
{
        uint8_t first[32];
        uint8_t second[32];
        size_t len = atoi(argv[1]);

        if (&second[0] - &first[0] < (std::ptrdiff_t)len)
                printf("too large\n");

        return 0;
}
```

PoC Application (*ARM* disassembly):
```
[...]
0x0040065e <+50>:    cmp     r2, r3
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
0x00400660 <+52>:    bge.n   0x40066c <main+64>
[...]
```

As seen in the disassembly, the *bge* instruction is used to branch and this is based on the flags set by *cmp*. The *bge* instruction branches depend on a signed compare, therefore negative values for *len* are capable of bypassing the check.

It is recommended to perform unsigned compares as *len* is actually of the *size_t* unsigned type. However, the cast turns the entire compare to a signed compare, which made this issue possible. Similarly to CT-01-005, the documentation only elaborates on one instance of this issue although it is recommended to review other areas of the code for similar patterns.

Note: *This issue was identified during a manual code audit but not yet verified on a running system. Given the time constraints of this assessment, the team opted for an increased coverage instead of clean reproducibility.*

CT-01-007 MCU: Buffer overflow via *Set*- and *Get*- attributes (*Critical*)

By abusing the issue described in CT-01-006, a buffer overflow vulnerability can be triggered through the *set_*- and *get_attributes()* endpoints. This potentially crashes the application due to the access of unmapped memory regions. More generally, it can lead to memory disclosure/corruptions.

What follows is an explanation how this memory corruption can be triggered, along with code excerpts showing the relevant pieces of code. The initial entrypoint is shown in the following part of the application's source code.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_server.c

Affected Code:

```
static hal_error_t pkey_get_attributes(const uint8_t **iptr, const uint8_t *
        const ilimit,uint8_t **optr, const uint8_t * const olimit)
{
[...]
    check(hal_xdr_decode_int(iptr, ilimit, &attributes_len));
[...]

    check(hal_rpc_pkey_get_attributes(pkey, attributes, attributes_len,
                                      attributes_buffer,
attributes_buffer_len));
[...]
                    err = hal_xdr_encode_variable_opaque(optr, olimit,
attributes[i].value, attributes[i].length);
```

A new attribute can be stored with a corrupted *length* value and *value* pointer, specifically using the bypass shown in CT-01-006. This corrupted attribute is then retrieved using the *get_attributes()* endpoint and the corresponding *pkey* handle. In

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

order to return the data, the attribute is serialized using the *hal_xdr_encode_variable_opaque()* function.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/xdr.c

Affected Code:
```
hal_error_t hal_xdr_encode_variable_opaque(uint8_t ** const outbuf, const
uint8_t * const limit, const uint8_t * const value, const size_t len)
{
    hal_error_t err;

    /* encode length */
    if ((err = hal_xdr_encode_int(outbuf, limit, (uint32_t)len)) == HAL_OK) {
        /* encode data */
        if ((err = hal_xdr_encode_fixed_opaque(outbuf, limit, value, len)) !=
HAL_OK)
[...]
hal_error_t hal_xdr_encode_fixed_opaque(uint8_t ** const outbuf, const uint8_t
* const limit, const uint8_t * const value, const size_t len)
{
[...]
    /* buffer overflow check */
    if (limit - *outbuf < (ptrdiff_t)((len + 3) & ~3))
        return HAL_ERROR_XDR_BUFFER_OVERFLOW;

    /* write the data */
    memcpy(*outbuf, value, len);
```

The corrupted *value* pointer and *length* field are passed down in the call chain and used to copy the data into the output buffer. This essentially boils down to a *memcpy()* call. Since the *len* parameter is either negative or very large when treated as unsigned, the *memcpy()* eventually overflows the *outbuf* buffer.

As in CT-01-006, it is recommended to perform an unsigned compare during the size check instead of manually casting to signed integers.

Note: *This issue was identified during a manual code audit but not yet verified on a running system. Given the time constraints of this assessment, the team opted for an increased coverage instead of clean reproducibility.*

CT-01-008 MCU: Session reuse via weak client handles *(High)*

A potential login bypass or session reuse vulnerability was found in the general mechanism that detects whether a user is authenticated against the RPC server at a given moment. The problem stems from the fact that only a 32-bit client handle is used inside the routine that checks whether a slot for this ID is allocated at the moment of the check. The login mechanism is shown in the following parts of the application's source code.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_server.c

Affected Code:
```
static hal_error_t login(const uint8_t **iptr, const uint8_t * const ilimit,
                         uint8_t **optr, const uint8_t * const olimit)
{
    hal_client_handle_t client;
    uint32_t user;
    const uint8_t *pin;
    size_t pin_len;

    check(hal_xdr_decode_int(iptr, ilimit, &client.handle));
    check(hal_xdr_decode_int(iptr, ilimit, &user));
    check(hal_xdr_decode_variable_opaque_ptr(iptr, ilimit, &pin, &pin_len));

    return hal_rpc_login(client, user, (const char * const)pin, pin_len);
}
```

One can observe that the client transmits a handle parameter that consists of a 32-bit integer value. By default, *cryptech_muxd* chooses a client handle that is dependant on the time, so it then increments on each incoming connection. In subsequent code, the client and its handle parameter are transparently handed over to the inner functions of the login code.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_misc.c

Affected Code:
```
static hal_error_t login(const hal_client_handle_t client,
                         const hal_user_t user,
                         const char * const pin, const size_t pin_len)
{
[...]
  if ((err = hal_pbkdf2(NULL, hal_hash_sha256, (const uint8_t *) pin, pin_len,
                        p->salt, sizeof(p->salt), buf, sizeof(buf),
iterations)) != HAL_OK)
    return err;

  unsigned diff = 0;
  for (size_t i = 0; i < sizeof(buf); i++)
    diff |= buf[i] ^ p->pin[i];

  if (diff != 0) {
    hal_sleep(HAL_PIN_DELAY_ON_FAILURE);
    return HAL_ERROR_PIN_INCORRECT;
  }

  return alloc_slot(client, user);
}
```

As depicted above, as soon as a matching *username/PIN* combination is found to exist in the current keystore, a slot is allocated to the authenticated. From then on, whenever a privileged operation verifies the presence of an authenticated user, it simply calls

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

hal_rpc_is_logged_in() to see whether the action is allowed or not. This can be observed on the items supplied next.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_server.c

Affected Code:
```
static hal_error_t is_logged_in(const uint8_t **iptr, const uint8_t * const
ilimit,
                                uint8_t **optr, const uint8_t * const olimit)
{
    hal_client_handle_t client;
    uint32_t user;

    check(hal_xdr_decode_int(iptr, ilimit, &client.handle));
    check(hal_xdr_decode_int(iptr, ilimit, &user));

    return hal_rpc_is_logged_in(client, user);
}
```

As one can see, the code again deserializes the integer value of the client handle and matches it against the currently allocated handles.

Affected File:
Source/Cryptech/releng/alpha/source/sw/libhal/rpc_misc.c

Affected Code:
```
static hal_error_t is_logged_in(const hal_client_handle_t client,
                                const hal_user_t user)
{
  if (user != HAL_USER_NORMAL && user != HAL_USER_SO && user != HAL_USER_WHEEL)
    return HAL_ERROR_IMPOSSIBLE;

  client_slot_t *slot = find_handle(client);

  if (slot == NULL || slot->logged_in != user)
    return HAL_ERROR_FORBIDDEN;

  return HAL_OK;
}
```

The problem with this approach is that an authenticated user will continue to use a predictable value for the client handle. Even a randomly chosen one would not be sufficient since it is easily possible to bruteforce a 32-bit value. As such, an attacker that is able to talk to the CrypTech USB device (depending on the system, the device file might be readable and writable), they can keep trying to send a handle to a client until it matches the currently allocated *hal_rpc_is_logged_in()* and returns true. This works as long as another authenticated user does not log out manually. Upon success, the attacker will have full privileges over a *wheel* or *so* user.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

It is recommended to implement a better session mechanism in which the client handle is chosen by the server. This item must be sufficiently large for making bruteforce attempts unfeasible.

CT-01-009 MCU: OOB read/write via ASN.1 key files with large numbers *(High)*

It was discovered that the function that parses ASN.1 keys does not check the size of the key's numbers. This makes it possible to corrupt memory and potentially execute arbitrary code by providing a key with large numbers. The bug can be triggered by importing a crafted key file. The issue was tested locally by providing a crafted key directly to the key parsing function at *hal_rsa_private_key_from_der()*.

Affected File:
libhal/rsa.c

Affected Code:

```
#define RSAPrivateKey_fields     \
  _(version);                    \
  _(key->n);                     \
[...]
  _(key->u);
[...]

hal_error_t hal_rsa_private_key_from_der(hal_rsa_key_t **key_,
                                         void *keybuf, const size_t keybuf_len,
                                         const uint8_t *der, const size_t
der_len)
{
  if (key_ == NULL || keybuf == NULL || keybuf_len < sizeof(hal_rsa_key_t) ||
der == NULL)
    return HAL_ERROR_BAD_ARGUMENTS;

  memset(keybuf, 0, keybuf_len);
  hal_rsa_key_t *key = keybuf;
[...]
#define _(x)                                                              \
  {                                                                       \
    size_t n;                                                             \
    err = hal_asn1_decode_integer(x, d, &n, vlen);                        \
    [...]                                                  \
  }
  RSAPrivateKey_fields;
#undef _
```

The parsed key is stored in the key structure which is a pointer to *keybuf* that needs to be a buffer with the size of the *hal_rsa_key_t* structure, with the latter being a fixed value. By using the *RSAPrivateKey_fields()*. the key parameters are read from the user-provided *der* buffer and stored in the *key* structure. The parsing that is relevant for this process happens in *hal_asn1_decode_integer()* where first the size of the value and then the value itself is obtained from.

Affected File:
libhal/asn1.c

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected Code:

```
hal_error_t hal_asn1_decode_integer(fp_int *bn, const uint8_t * const der,
size_t *der_len, const size_t der_max)
{
[...]
    if ((err = hal_asn1_decode_header(ASN1_INTEGER, der, der_max, &hlen,
&vlen)) != HAL_OK)
    return err;
[...]
  fp_init(bn);
  fp_read_unsigned_bin(bn, (uint8_t *) der + hlen, vlen);
  return HAL_OK;
}
```

The *fp_read_unsigned_bin()* function reads a number from a buffer and stores it in the appropriate *fp_int* variable pointed to by *bn*. Different code sections are used based on the system's endianness.

Affected File:
/sw/thirdparty/libtfm/tomsfastmath/src/bin/fp_read_unsigned_bin.c

Affected Code:

```
void fp_read_unsigned_bin(fp_int *a, const unsigned char *b, int c)
{
  /* zero the int */
  fp_zero (a);
#if (defined(ENDIAN_LITTLE) || defined(ENDIAN_BIG)) && !defined(FP_64BIT)
 [...]
  {
     unsigned char *pd = (unsigned char *)a->dp;

     if ((unsigned)c > (FP_SIZE * sizeof(fp_digit))) {
        int excess = c - (FP_SIZE * sizeof(fp_digit));
        c -= excess;
        b += excess;
     }

[...]
#else
  /* read the bytes in */
  for (; c > 0; c--) {
     fp_mul_2d (a, 8, a);
     a->dp[0] |= *b++;
     a->used += 1;
  }
#endif
```

It is assumed that one of the *ENDIAN_LITTLE* or *ENDIAN_BIG* pair is always defined, but in case *FP_64BIT* is defined as well, the length in *c* is not checked and *a->used* can grow to a value that exceeds the size of *a->dp*. In turn, this results in an OOB read/write in *fp_mul_2d()*.

Affected File:
/sw/thirdparty/libtfm/tomsfastmath/src/mul/fp_mul_2d.c

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected Code:

```
for (x = 0; x < c->used; x++) {
        carrytmp = c->dp[x] >> shift;
        c->dp[x] = (c->dp[x] << b) + carry;
        carry = carrytmp;
}
```

The core problem here is that the for-loop in *fp_read_unsigned_bin()* can increase *a->used* indefinitely, thus causing OOB memory access in *fp_mul_2d*. Therefore, it is recommended to always execute the length check in *fp_read_unsigned_bin()*.

Note: *This issue was identified during a manual code audit but not yet verified on a running system. Given the time constraints of this assessment, the team opted for an increased coverage instead of clean reproducibility.*

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

CT-01-001 FPGA: Incomplete asynchronous reset of FPGA modules *(Low)*

The FPGA design features an asynchronous reset input PIN. Asserting the reset signal cancels all running operations and restores the circuit's initial state. Although it is not strictly required to reset each and every bit, it remains strongly recommended. Firstly, it is because resetting all registers guarantees to completely restore the initial state and prevents error conditions from being carried over. Secondly, clearing all stored data makes sure that information like plaintext or key material is completely wiped and can no longer be accessed.

Affected File:
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes_key_mem.v

Affected Code:

```
  reg [127 : 0] key_mem [0 : 14];
  [...]
  reg [127 : 0] prev_key0_reg;
  [...]
  reg [127 : 0] prev_key1_reg;
  [...]
  always @ (posedge clk or negedge reset_n)
    begin: reg_update
      [...]
      if (!reset_n)
        begin
          for (i = 0 ; i < 4 ; i = i + 1)
            key_mem [i] <= 128'h0;

          rcon_reg        <= 8'h0;
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
        ready_reg       <= 1'b0;
        round_ctr_reg   <= 4'h0;
        key_mem_ctrl_reg <= CTRL_IDLE;
      end
      [...]
  end // reg_update
```

The *aes_key_mem* module generating AES round keys does not clear registers, specifically *prev_key0_reg[]* and *prev_key1_reg[]*, while it also preserves *key_mem[4:14]*. Although a reset *prev_key0_reg[]* and *prev_key1_reg[]* are always written before read, *key_mem[]* can be accessed by the parent modules and even used by the microcontroller without initialization.

Note 1: *The code furnished here is just one example of this behavior. It is further recommended to review other modules and establish a design pattern where all registers are cleared upon reset.*

Note 2: *Block RAM as instantiated in* ECDSA *and* MODEXP *modules cannot be reset asynchronously but may be replaced with distributed RAM.*

CT-01-002 FPGA: Side-effects on writes to memory-mapped interface *(Low)*

Each FPGA core provides a memory-mapped interface. Usually the core's parameters are set by writing values into an appropriate configuration registers before input data is copied and the processes are started by setting specific bits of control registers.

Some cores have undocumented restrictions on using the memory-mapped interface. In many cases, it makes no sense to write certain values or change registers at specific times. But doing so anyway, this is not always handled correctly and can cause side-effects that result in invalid output.

Affected Files:
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes.v
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes_core.v

In the AES module, the round keys are updated by writing the *INIT* bit of the *CTRL* register. This can be done simultaneously to a running en- or decryption, thereby changing the key halfway through processing. Writing to the *CONFIG* register is also possible at any time and changes to the key length or mode are immediately effective. Depending on the timing, completing such action during a running process very likely corrupts the output.

Note: *The above listed weaknesses are just examples. It is further recommended to review other cores and restrict writing as much as possible by either ignoring unacceptable input or latching values during running processes.*

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

CT-01-003 FPGA: Readout of intermediate results via the external Interface *(Low)*

The cores provide their output via registers of the memory-mapped interface. They assert a valid flag once the results are ready. Some cores use their output registers during calculations and thereby allow reading of intermediate results.

Affected Files:
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes.v
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes_core.v
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes_decipher_block.v
Source/Cryptech/releng/alpha/source/core/cipher/aes/src/rtl/aes_encipher_block.v

In the AES core, none of the modules mask the output before it is exposed to the memory-mapped interface and, thus, to the microcontroller. Reading registers *RESULT0* to *RESULT3* always returns the values of the currently processed round, possibly exposing key material.

Note: *The above listed weaknesses are just examples. It is further recommended to review other cores and restrict reading of results until they are actually ready.*

CT-01-004 MCU: Hardening memory-mapped interface access in *libhal* *(Low)*

Within the microcontroller, *libhal* provides low-level drivers for accessing the FPGA cores' functionality. Some functions interacting with the FPGA cores do not check for the *ready* flag before writing registers or starting a process. At times, in addition, the *ready* flag is checked instead of the *valid* flag before reading results.

Affected File:
*Source/*Cryptech/releng/alpha/source/sw/libhal/aes_keywrap.c

Affected Code:
```
static hal_error_t do_block(const hal_core_t *core, uint8_t *b1, uint8_t *b2)
{
  hal_error_t err;

  hal_assert(b1 != NULL && b2 != NULL);

  if ((err = hal_io_write(core, AES_ADDR_BLOCK0, b1, 8)) != HAL_OK ||
      (err = hal_io_write(core, AES_ADDR_BLOCK2, b2, 8)) != HAL_OK ||
      (err = hal_io_next(core))                          != HAL_OK ||
      (err = hal_io_wait_ready(core))                    != HAL_OK ||
      (err = hal_io_read(core, AES_ADDR_RESULT0, b1, 8)) != HAL_OK ||
      (err = hal_io_read(core, AES_ADDR_RESULT2, b2, 8)) != HAL_OK)
      return err;

  return HAL_OK;
}
```

The *do_block()* function is called directly after loading a key and the *ready* flag is not verified before starting a new en- or decryption. Furthermore the *ready* flag is checked instead of the *valid* flag prior to the output's reading. This results in a race condition

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

which, due to the system's current timing characteristics, is unlikely to cause any actual problems.

Note: *The above listed code excerpt is just one example. It is strongly recommended to review other modules and ensure the correct use of the memory-mapped interface for the cores.*

Conclusions

All in all, the results of this September 2018 security assessment of the CrypTech/DiamondKey HSM scope are suboptimal. The testers from both Cure53 and Trustworks can only conclude that the project, in its current state, calls for considerably more security attention. This can be drawn from the underwhelming quality of the examined code, as well as clearly stems from the extensive number of five issues considered to be of either "*Critical*" or "*High*"-severity. In other words, the tested compound is at an increased risk of a compromise and presently exhibits limited success when it comes to meeting security goals.

As this white-box method assessment involved eight testers who spent 32 days on examining various items in scope under the premise of six distinct work packages, the following sections will reflect on the detailed WP outcomes in a more structured fashion.

C-Code Audit

The code auditors are in agreement that the quality of the C code can definitely be improved. One positive takeaway, however, is that the spotted flaws stem from a single issue, which is seemingly rather easy to fix. For example, the problem noted in CT-01-006 describes a bypass that can happen due to a misconception about the *length* check concerning opaque pointers. With the removal of the signed cast and the resulting fields with negative lengths, the tested compound became exposed to a number of resulting issues. To reiterate, these can luckily be rectified in one sweep. Next up, the testers unveiled an entire, highly prominent bug class. The issue, documented as CT-01-005 shows that dynamic stack allocations are entirely user-controlled and may offer arbitrary read/writes in the worst-case scenario. As this behavior has been observed at multiple and different locations in the code base, it is necessary to develop a comprehensive fix and ensure that all occurrences are actually encompassed by the repairs. Once these two broader patterns of attacks can be prevented, it is believed that the already present sanity checks will be able catch most of the buffer overflow and signed/unsigned vulnerabilities.

However, there are still weaknesses that demonstrate that bad design decisions were made at one point or another. One piece of evidence for this is the fact that the login-check is not centralized. This means that multiple RPC calls have to be made in hopes of manually verifying whether a given current user has actually been authenticated. With this in mind, it is understandable that forgetting to add the actual *hal_rpc_is_logged_in()*

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

check can be quite common during the process of establishing new RPC calls. Henceforth, unauthenticated users have the power to issue privileged commands. Once more, the problem could be eradicated in one go if a more centralized approach became the new standard.

Staying on the topic of incorrect design choices, the problem discussed in CT-01-008 illuminates that the client's session (actually called the client handle) is not set by the server but rather chosen by the client. Badly written clients can thus use a weak session-ID and risk their session getting hijacked by attackers that simply wait for another user to log in and effectuate a takeover afterwards. Moreover, also the server components need to make sure this handle is not trivial to brute-force.

Last but not least, CT-01-009 is a result of a parsing vulnerability for the ASN.1 keys. Since this issue was actually found through fuzzing, it is important that also any third-party code is extensively tested before being shipped to sensitive hardware such as CrypTech/DiamondKey. Although test-cases for unit-testing are present, it would be fruitful to combine them with fuzzing frameworks, so that the incorrectly formed keys can be encompassed by the prevention mechanisms too. This would actually be a vital decision because malicious public keys (i.e. able to exploit a device) signify the worst-case scenario for the integrity of the tested compound. In that sense, parsing needs to be extremely robust and defensive, absolutely guaranteeing that all incorrectly formatted fields are consistently handled in a proper way.

Cryptography

In this part of the assessment, all of the specifications were reviewed in full and in a detailed manner. Importantly, the code connected to the primitives - AES, hash functions, ChaCha, ECDSA - was also reviewed. Completing the items under investigation, some of the custom, internal protocols such as *cryptech_backup*, were also included in the assessment.

It needs to be mentioned that the cryptographic part of the project could have benefitted from a slightly better information exchange right at the gate. The provided specifications, namely "*Alpha Developer Reference*" and also the *TRNG* architecture, were extremely helpful and well-written. It is thus very surprising that they were not originally included with the materials received by the testers. It is hard to find a reason behind this, as this omission cost the project valuable time early on.

Having said that, the results in the cryptographic realm are outstandingly positive. Not only there were no security issues found, but also the overall design has been evaluated as excellent. This especially holds for the *TRNG,* which displays many strengths despite its simple architecture. The testing team is happy to report that the cryptographic aspects connected to the tested items are well under-control.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Verilog Code Audit

While certain security goals have been accomplished by the Verilog code, its audit led to the unveiling of some minor problems. The FPGA is generally well-designed and correctly implemented. All modules have been written in a straightforward manner, making the code easy to read and audit for security issues. The concept of having individual and independent cores that share a memory-mapped interface in a well-defined manner, makes the design scalable. With this in place, adding new cryptographic modules should not affect previously established and tested modules. The code has enough comments for orientation, yet it would be advised for the modules' interfaces and the restrictions on using them to be documented. On a plus side, there are plenty of test-benches for simulation and example drivers for the STM microcontroller that demonstrate the intended use are available.

In sum, the review of the Verilog code did not reveal any exploitable vulnerabilities but simply let the testing team highlight some issues that could lead to problems in the future. It should be noted that all of the listed flaws and points were documented with examples from the AES core because this core was the first one to be reviewed and featured all types of the uncovered problems. It needs to be clearly understood that this does not make it the only affected core.

Among the weaknesses, a reset signal (see CT-01-001) within a hardware design should probably reset as many registers as possible. The few additional routing resources that are needed can probably be seen as negligible for the FPGA. Being able to wipe key material could come in handy when reacting to tampering with the device. Similarly, from a security perspective it would be a good idea to make the interface between FPGA and microcontroller as robust as possible. Connected to this, CT-01-002 lists some examples for input that might not be expected but can nevertheless cause strange or otherwise undesirable behavior.

Next, short notes must be made for the fact that intermediate results should not be exposed for cryptographic operations without a very good justification (see CT-01-003) and the implementation of *libhal* could take a little more care when it comes to checking flags (see CT-01-004). While *ready* and *valid* may currently have the same meaning, the robustness principle also applies to the Microcontroller. Observations and evidence collected during the assessment points to a suggestion for the FPGA to offer some means of software resetting in connection to individual cores. The Microcontroller's *libhal* could perform a reset after loading and before unloading a core, in doing so clearing the internal states and wiping the key material. Furthermore, it is suggested that the Microcontroller could implement a type of an execution-time watchdog. In other words, an operation could be cancelled and the FPGA consequently reset if the result is not available within a timeout, thus eliminating risks linked to the Denial-of-Service (DoS) issues.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

General Impressions

This September 2018 joint security project executed by Cure53 and Trustworks is quite significant because of its vitally important subject matter. Specifically, the CrypTech/DiamondKey open source HSM project sets out to allow the community to secure important Internet components like i.e. BGP via RPKI, as well as others. It is pivotal that the CrypTech/DiamondKey compound seeks to ensure transparency and cryptographic security to the users. As the goals are quite ambitious and sensitive, the results must be interpreted with due diligence. This means that even a theoretically low-number of nine security-relevant discoveries must be seen in the context of their increased severities and the paramount risks that they carry.

To reiterate some details, this project proceeded swiftly, since unencumbered access to all source code, documentation and open communication with the developers was granted. Three test devices were delivered to two different teams, one of which was made remotely accessible to verify assumptions made during the audit. While a complete pre-built firmware-bundle was provided for the convenience of the auditors and used for the best part of the assessment, a custom firmware-bundle was also created and flashed in order to better understand the deployment process in its entirety.

Eight members of the Cure53 & Trustworks testing team believe to have reached a complete coverage and gained in-depth familiarity with the inner-workings of the hardware/firmware combination. Their investigations spanned the client-side through to the MCU and ending in the FPGA, which was made accessible via the extended design documents and additional consultations with the developers. An exception regarding coverage entailed not attempting attacks against the circuit board and its tamper-proofing of the individual components, along with thermal side-channel attacks. This is because the current system, while quite complete and in good shape, is still at the Alpha phase of development. Consequently, this aspect was simply left out of scope at this stage.

All six Work Packages were completed in a timely manner, beginning with the review of documentation (WP1), then building the firmware and booting the system (WP2), all underpinning the key auditing of the cryptographic and security concepts, the general C code audit together with the verification of the interface code, and the Verilog implementation of the underlying FPGA-based hardware (WPs 3-6).

The testing team would like to express that particular effort was invested into the theoretical and practical verification of all supported cryptographic primitives like AES, ChaCha, SHA1/2 and internal protocols like *cryptech_backup*. The implementation of the individual primitives, in detail AES, ChaCha, SHA1, SHA256/512, ECDSA256/384, MODEXPA7 and TRNG was extensively probed for illicit states and results that could potentially expose plaintext or key material, including the search for possible time-based side-channel attacks. The cryptographic design deployed by the CrypTech/DiamondKey compound deserves highest praise.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Closing Notes

To conclude, the current state of security observed at the CrypTech/DiamondKey HSM firmware project is no small feat. The testers acknowledge the accomplishments and are convinced that the development team needs to be congratulated and celebrated for reaching this important milestone of being subjected to external scrutiny. The vulnerabilities isolated during the this assignment, while mostly easy to fix or at least not overly difficult in terms of devising appropriate hardening mechanisms, nevertheless clearly show that additional revisions and attention are necessitated by several aspects of the analyzed HSM-implementation. It is recommended that the entire project is audited by the development team in accordance with the notes provided by the auditors, as only such an extensive and all-encompassing approach can make it possible for the project to finally leave the Alpha-stage and do so as a much more secure compound. Once the issues are fixed and the design decisions reevaluated, Cure53 and Trustworks believe that a state of maturity and readiness for general deployment can be reached. This is considered a necessity because of the single-pointed significance of the HSM in a truly secure cryptographic environment.

Cure53 would like to thank Phil Roberts, Jakob Schlyter, Dominique Douglas, Joachim Strömbergson and Rob Austein from the respective CrypTech and DiamondKey teams for their excellent project coordination, support and assistance, both before and during this assignment.